Efficient and Robust Dynamic Scheduling and Synchronization in Practical Embedded Real-Time Multiprocessor Systems

Dissertation

zur Erlangung des Doktorgrades der Naturwissenschaften

vorgelegt von

Martin Alfranseder

aus Eggenfelden

genehmigt von der Fakultät für Mathematik/Informatik und Maschinenbau
der Technischen Universität Clausthal,

Tag der mündlichen Prüfung

16.09.2016

Dekan

Prof. Dr. Jürgen Dix

Vorsitzender der Prüfungskommission

Prof. Dr. Volker Wesling

Betreuer

Prof. Dr. Christian Siemers

Gutachter

Prof. Dr. Andreas Rausch

Prof. Dr. Jürgen Mottok

For Eva

Kurzzusammenfassung

Multi-Prozessor Systeme haben in eingebettete Systeme mit Echtzeit Anforderungen Einzug gehalten. Für Echtzeitsysteme ist entscheidend, dass sowohl funktionale als auch temporale Erfordernisse eingehalten werden. Diese Dissertation beschäftigt sich mit dem Aspekt der zeitlichen Anforderungen an Echtzeitsysteme. In diesem Kontext stehen zwei grundlegende Fragen im Raum, nämlich: (1) Wie verteilt man die zur Ausführung bereiten Tasks am besten auf die zur Verfügung stehenden Rechenkerne? (2) Wie synchronisiert man die Tasks ohne dass die Datenkonsistenz beeinträchtigt wird und unnötige oder endlose Blockierzeiten entstehen? Es wurden bereits Scheduling-Algorithmen entwickelt, die theoretisch optimal sind. Optimal bedeutet in diesem Kontext, dass wenn immer es eine Möglichkeit gibt, ein Task-Set ohne Deadline Verletzung auszuführen, der Algorithmus eine solche Möglichkeit auch findet. Diese optimalen Algorithmen haben sich jedoch in der Praxis (z.B. im Antriebsstrang von Automobilen) bisher nicht durchgesetzt, da mit dem Einsatz dieser Scheduling Algorithmen eine Vielzahl an Kern-Migrationen, Kontextwechsel und die Notwendigkeit von Prioritäts-Neuberechnungen einhergehen, die zu Verzögerungen des gesamten Systems führen. Diese Arbeit versucht, die Lücke zwischen Theorie und Praxis ein Stück weit zu schließen. Dafür wurde ein Konzept für einen globalen Scheduling Algorithmus entwickelt, der effizient zu implementieren ist und Prioritätsänderungen während der Laufzeit zulässt. Er enthält außerdem Erweiterungen, die beim Einsatz in Motorsteuergeräten benötigt werden. Im Fall der zweiten Frage stellte sich heraus, dass es kein Synchronisationsprotokoll gibt, das für alle Arten von Task-Sets und Hardwarearchitekturen optimal ist, also so wenig Blockierzeiten wie möglich verursacht. Daher werden im Rahmen dieser Arbeit zwei Synchronisationsprotokolle für Multi-Prozessor Echtzeitsysteme vorgestellt, die bisher entwickelten Protokollen in der Vermeidung von Blockierzeiten für bestimmte Taks-sets überlegen sind. Diese Protokolle werden mittels diskreter, event-basierter Simulation im Rahmen einer Fallstudie experimentell evaluiert.

Abstract

Multiprocessors are on the way to become a standard platform for embedded systems with real-time requirements. For real-time systems, it is crucial that functional and temporal constraints are fulfilled. The focus of this dissertation lies on the aspect of these timing requirements. Within this context, two fundamental questions arise: (1) How to schedule a given task set on the available cores; and (2) How to synchronize tasks without a loss of data consistency and unbounded blocking times. In theory, optimal scheduling algorithms are already known. In this case, optimal means that if there is any option that a task set fulfills all its temporal requirements, the algorithm finds such a schedule. Nevertheless, they are not in practical use (at least in automotive power-train systems) because these optimal algorithms are accompanied by overheads caused by core migrations, context switches and the need for repeatedly priority calculations. This thesis tries to close the gap between theory and practice by developing a concept for an efficient implementation global scheduler with dynamic priorities that fulfills the requirements of engine control units. Furthermore, there exists no generic optimal multiprocessor real-time synchronization mechanism that is able to handle all types of task allocation and priorities, and supports as well nesting of resource requests. Thus, two different locking protocols are provided that outperform existing ones for specific types of task sets for embedded multiprocessor real-time systems. Case studies are presented that evaluate the assumptions of the efficiency and robustness of the locking protocols.

Acknowledgements

First of all, I would like to thank my advisor, Prof. Dr. Jürgen Mottok who taught me how to resarch. Without his warm guidance and persistent help this dissertation would not have been possible. I am deeply grateful to my Prof. Dr. Christian Siemers for his great support and fruitful discussions. He made it possible for me to write my thesis at his research group at the Technical University of Clausthal.

This work arises from the research project S^{3}Core, which is funded by the Bayerische Forschungsstiftung BFS with the title 'Entwicklung von **S**cheduling Verfahren und Kommunikationsmechanismen für **S**icherheitskritische Multi**core** Echtzeit-**S**ysteme, sowie Verfahren zur Analyse und Bewertung deren Echtzeitanforderungen'. S^{3}Core is a cooperative research project between the Laboratory for Safe and Secure Systems LaS^3 which is part of the Ostbayerische Technische Hochschule Regensburg, Continental Automotive GmbH, Audi AG, Timing Architects Embedded Systems GmbH, TÜV Süd Automotive GmbH, Intence Automotive Electronics GmbH, Friedrich-Alexander Universität Erlangen-Nürnberg and Technische Universität Clausthal.

Special thanks to all the colleagues of Continental and Timing Architects for helping me out several times and for the great time we had. Further, I would particularly like to thank my advisors from Continental, Ralph Mader and Prof. Dr. Michael Niemetz for many inspiring and illuminating discussions. I have greatly benefited from the periodically research meetings with my advisor at Timing Architects, Dr. Michael Deubzer. I would like to thank my research colleague Matthias Mucha for the great time we had at the LaS^3, for the daily beneficial scientific exchange, shared sorrows ;) and jokes. Of course I would like to thank my other colleagues from the LaS^3, in particular Stefan Krämer for many hours of debugging the simulator and answering my annoying questions, Stefan Schmidhuber, Andreas Sailer and Jürgen Braun. Furthermore, I want to thank the students I have advised, Tobias Krapf and Markus Daub for their support in developing the scheduler and the locking protocols.

I am deeply thankful to have such a wonderful familiy. My parents, Marianne and Gerhard have always trusted in me and gave me all their support and patience I needed to be able to finish this work. Special thanks go to my brother Michael and my sisters Veronika and Elisabeth, for all their support and encouragement, and for helping me to calm down from work many times. Finally, I would like to thank my wife Eva for her love and understanding, and for following me to Regensburg. You always supported me with great patience, encouragement and unwavering love. You improved my English significantly, without you this thesis would be only half as comprehensible. You enrich my life every day and always bring a smile to my face.

Contents

List of Figures

List of Tables

Abbreviations

AUTOSAR	**AUT**omotive **O**pen **S**ystem **AR**chitecture
BF	**B**oundary **F**air
BIP	**B**andwidth **I**nheritance **P**rotocol
CAN	**C**ontroller **A**rea **N**etwork
CO2	**C**arbon Di-**O**xyde
CPU	**C**entral **P**rocessing **U**nit
CS	**C**ritical **S**ection
DES	**D**iscrete **E**vent **S**imulation
DM	**D**eadline **M**onotonic
DM-PM	**D**eadline **M**onotonic with **P**riority **M**igration
D-PCP	**D**istributed **P**riority **C**eiling **P**rotocol
EDDP	**E**arliest **D**eadline **D**eferrable **P**ortion
EDF	**E**arliest **D**eadline **F**irst
EKG	**E**DF with task splitting and **K** processors in a **G**roup
EPDF	**E**arliest **P**seudo-**D**eadline **F**irst
ERfair	**E**arly **R**elease **Fair**
FEP	**F**orced **E**xecution **P**rotocol
FIFO	*F*irst **I**n, **F**irst **O**ut
FMLP	**F**lexible **M**ultiprocessor **L**ocking **P**rotocol
FMLP+	**F**IFO **M**ultiprocessor **L**ocking **P**rotocol
GE	**G**ross **E**xecution **T**ime
G-EDF	**G**lobal **EDF**
GFL	**G**lobal **F**air **L**ateness
GPU	**G**raphics **P**rocessing **U**nit
HLP	**H**ighest **L**ocker **P**rotocol

LLF	**L**east **L**axity **F**irst
LLREF	**L**east **L**ocal **R**emaining **E**xecution Time **F**irst
MAST	**M**odeling and **A**nalysis **S**uite for Real-**T**ime Applications
M-BWI	**M**ultiprocessor **B**andwidth **I**nheritance **P**rotocol
M-DPCP	**M**ultiprocessor **D**ynamic **P**riority **C**eiling **P**rotocol
M-PCP	**M**ultiprocessor **P**riority **C**eiling **P**rotocol
MSOS	**M**ultiprocessor **S**ynchronization Protocol For Real-Time **O**pen **S**ystems
MrsP	**M**ultiprocessor **R**esource **S**haring **P**rotocol
M-SRP	**M**ultiprocessor **S**tack **R**esource **P**olicy
NE	**N**etto **E**xecution Time
NPP	**N**on-**P**reemptive **P**rotocol
NRT	**N**ormalized **R**esponse **T**ime
NUMA	**N**on-**U**niform **M**emory **A**ccess
O	Upper bound on the growth rate of the function
OPA	**O**ptimal **P**riority **A**ssignment
OMLP	**O(m) L**ocking **P**rotocol
OS	**O**perating **S**ystem
OSEK	**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
PCP	**P**riority **C**eiling **P**rotocol
P-EDF	**P**artitioned **EDF**
Pfair	**P**roportionate **fair**
PIP	**P**riority **I**nheritance **P**rotocol
PLWP	**P**reemptable **W**aiting **L**ocking **P**rotocol
RAM	**R**andom **A**ccess **M**emory
RM	**R**ate **M**onotonic
RNLP	**R**eal-Time **N**ested **L**ocking **P**rotocol
RPC	**R**emote **P**rocedure **C**all
RT	**R**esponse **T**ime
RTA	**R**esponse **T**ime **A**nalysis
RTC	**R**eal-**T**ime **C**alculus
RTOS	**R**eal-**T**ime **O**perating **S**ystem
RUN	**R**eduction to **UN**iprocessor
SMP	**S**ymmetric **M**ulti**P**rocessors

SRP	**S**tack **R**esource **P**olicy
TA	**T**imed **A**utomata
UMA	**U**niform **M**emory **A**ccess
WCRT	**W**orst **C**ase **R**esponse **T**ime

Chapter 1

Introduction

Multiprocessor systems have become a common approach in many applications nowadays. Many-core platforms like the ones from Tilera [134] (up to 100 cores) and Kalray (256 cores) [86] (network virtualization) or general purpose computing with graphics processing units (GPUs) (nVidia [50], 192 cores) are used for video stream processing, machine learning and much more (see catalog [51]).

However, in some fields of embedded real-time systems, multiprocessors are less frequent because of their different (strict) requirements. Nevertheless, there already exist multicore micro-controllers, for example in the automotive domain, as there are the Infineon AURIX family [133] or the Freescale MPC5676R [124], equipped with 3 respectively 2 cores. The reason for the change to multiprocessor systems lies inter alia in the need for additional functionality in those systems. For instance, in an automotive powertrain system additional functionality is required to reduce fuel consumption and CO_2 emissions. Further examples are driver assistance functionalities. For single-core processors this means that the processor frequency has to be raised, since nowadays processor utilizations from 80 to 95 % are usual in automotive powertrain systems [109].

However, this goes ahead with several disadvantages. First, energy consumption increases quadratic related to the raise of frequency. Further, heat dissipation and electromagnetic perturbations increase as well. The advent of multiprocessors overcome this problems, as processing capacity can be added without rising the frequency. Todays automotive powertrain multiprocessor applications use partitioned task-fix priority scheduling, where scheduling algorithms are that ones from single-core applications (e.

g. OSEK- [116] or AUTOSAR scheduling [14]). This kind of scheduling still lets some gaps in terms of robustness and efficiency of those systems. First, task-fix priority is not even optimal for single-core processors, as for instance [101] showed for sporadic task sets with implicit deadlines. Additionally, partitioned scheduling requires a kind of task-to-core assignment, which is often realized by a kind of bin-packing. The lack of optimality for this kind of scheduling can be seen in [105] where utilization bounds for partitioned, rate monotonic scheduling are provided, with the restriction that these bounds are only valid for periodic task sets. There already exist (theoretical) optimal scheduling algorithms, although they are not in practical use because of their overhead (many context switches, migrations and re-calculations of priorities, see also [29]).

Goal of this thesis is to optimize temporal *robustness* and *efficiency* of embedded multiprocessor real-time systems. *Robustness* describes, how a real-time system is capable to fulfill its temporal requirements despite perturbations (e. g. synchronization overhead) while *efficiency* refers to different aspects, as there were a well-balanced core utilization and low implementation and processing overheads. Kopetz and Baruah defined in [93] and [19] respectively that a robust system keeps its schedulability even when it operates beyond the worst-case assumptions used in its schedulability test. Note, that robustness is limited, since each system will fail regarding its temporal constraints at a certain point of temporal perturbations [19].

However, to reach this goals robust and efficient scheduling algorithms and synchronization mechanisms are required. This dissertation provides a concept of a scheduler for embedded multiprocessor real-time systems, that rises the robustness and efficiency of those systems in contrast to existing approaches. Further, two locking protocols are presented, that make multiprocessor synchronization more robust efficient for certain real-time task sets. The term *efficient* here refers to a reduction of blocking times compared to existing approaches which leads to a higher robustness. The following sections give an overview of the contribution of this thesis as well as of the thesis structure.

1.1 Contribution

The contributions of this dissertation are twofold: The first part of the contribution addresses the question about how efficient and robust scheduling of task sets with multiple time base activations on a multiprocessor with a given operating system can be realized.

Robust here refers to the requirement of real-time systems that temporal constraints (deadlines) have to be kept, even if the defined boundary conditions are violated. The term efficient contains two different meanings. On the one hand, an efficient scheduler leads to a well-balanced core utilization of the multiprocessor system. On the other hand, it requires only low implementation and scheduling overhead.

We present a scheduler for complex embedded multiprocessor real-time systems. Global task allocation with job-level priorities is applied. The contributed scheduler further runs on a existing platform with partitioned allocation and task-fix priorities, which goes along with the advantage that already existing (and well analyzed and tested) features like task handling (saving the task context, manage dispatching and termination etc.) can be reused. The presented scheduler further provides opportunities that special requirements from engine management systems in the field of automotive powertrain systems are supported, as there were system state transitions, data buffering or chained tasks etc.

The second part of the contribution discusses efficient multiprocessor synchronization. Efficient in this context means that unnecessary blocking times, especially for high priority tasks, are avoided as far as possible. Note that efficient synchronization mechanisms help to optimize robustness of real-time systems as it is defined above in this section. Two different locking protocols are presented, both are based on FIFO sorted wait-queues. Furthermore, both protocols support partitioned, global and clustered allocation as well as all types of prioritization and further the nesting of resource requests. The preemptable waiting locking protocol PLWP uses preemptive busy-waiting and non-preemptive critical sections. In contrast, the forced execution protocol FEP remains also preemptive within critical sections and ensures progress by a priority boosting mechanism.

1.2 Thesis Structure

The remainder of this work is structured in the following chapters. In Chapter 2 the definitions and assumptions regarding hardware and software architecture are described on which the dissertation is based. Chapter 3 provides the discussion of related work about real-time multiprocessor scheduling and multiprocessor real-time locking protocols. The contributions of the dissertation are provided in Chapter 4. A concept for

a global scheduler with job-level fix priority scheduling is presented which can be implemented on an existing partitioned scheduler with task-fix priorities. Further, two different locking protocols are presented that are able to reduce blocking time and priority inversions. Furthermore, Chapter 5 presents different case studies where the locking protocols are evaluated regarding the aspect of what kind of task set which one performs best. Finally, in chapter 6 the results are summarized and future work is discussed.

Chapter 2

Definitions

Within this chapter the fundamentals of the thesis are introduced and definitions are given about all topics that correlate with the contribution. After a generic definition of embedded real-time systems, the specific multiprocessor hardware and -software are discussed. Further, a definition of resources is given as well as a formulation of the scheduling problem and basic definitions about real-time examination.

2.1 Embedded Real-Time Systems

First, we define the term embedded real-time systems as it is used in the scope of this thesis. We start with the definition of an embedded system:

Definition 2.1. An embedded system consists of hardware and software components and is embedded in a complex technical environment.

Real-time systems calculate a new output as a reaction of events from their environment. Further, they have to guarantee that correctness and deadlines of the calculated results are kept. The focus of this thesis lies in real-time computing systems, so the definition of Kopetz can be applied:

Definition 2.2. 'A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.' [92]

If a real-time system produces a correct result after its deadline it does not fulfill its timing constraints and thus the result is not valid. Buttazzo classifies real-time tasks into three different categories (*criticality*), depending on the consequences that occur if a deadline is missed [36]:

- '**Hard**: A real-time task is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under control.
- **Firm**: A real-time task is said to be firm if producing the results after its deadline is useless for the system, but does not cause any damage.
- **Soft**: A real-time task is said to be soft if producing the results after its deadline has still some utility for the system, although causing a performance degradation.'[36]

An embedded real-time system may contain tasks of different categories of the upper definition. Such systems are called systems with *mixed criticality*.

Abstract models are used to describe the different hardware and software components that build an embedded real-time system in combination. In the following sections, the embedded real-time system model is introduced. It describes all necessary properties of embedded multicore real-time systems regarding the scope of this work.

2.2 Multiprocessor Hardware

In this section a short introduction considering the different types of multiprocessors is given. In terms of computer architecture, a *multiprocessor* (*multicore processor*) is a system that consists of multiple and independently processing units (on a single chip) that communicate via a processor interconnect [132]. Regarding the processor interconnect, one can distinguish between two classes of multiprocessors. *Shared-memory multiprocessors* have a common central memory that can be accessed by all processors, and the processors are connected to each other. In contrast, *distributed-memory multiprocessors* (*multicomputers*) are hardware systems where each processor has its own local memory which can not be accessed by the other processors. The processors of distributed-memory multiprocessors are still connected to each other, but only by a *message bus*.

Shared-memory multiprocessors further differ according memory access patterns and processor symmetry. *Symmetric multiprocessors* (SMPs) consist of a number of identical processors, while in *uniform heterogeneous multiprocessors* processors differ in processing speed only. In contrast, *unrelated heterogeneous multiprocessors* contain processors that may have different special capabilities each.

Systems where a processor is able to access each memory location within the same maximum latency are called *uniform memory access* (UMA) architectures [31]. Different to this, systems where the available memory is split across several modules where some of them are closer to a particular processor than others are denoted a *non-uniform memory access* (NUMA) architecture.

In this work we focus on shared-memory multiprocessors that are built on a single integrated circle chip. This can also be named *multicore* design. In general, we assume to deal with symmetric multiprocessors with uniform memory accesses except it is explicitly defined differently.

2.3 Multiprocessor Software

An embedded real-time system only fulfills its purpose if both, logical and temporal constraints, are obeyed. Logical constraints are fulfilled by software functions that calculate a new system output based on input data from the environment. In contrast, to fulfill the temporal requirements, these software functions are divided into real-time tasks. The collection of all tasks T_i (i = 1,...,n) in a real-time system S is called task set $\tau = \{T_1, ...T_n\}$ of S. In the following sections, the software system is described more in detail and contains descriptions and definitions about tasks, jobs and runnables that together build the software system that is used in the context of this thesis.

2.3.1 Tasks

In general, a task consists of sequential pieces of software. A task has to be invoked to fulfill the logical and functional requirements of the real-time system it belongs to. A real-time task is defined by its timing parameters.

Definition 2.3. Each task T_i contains the characterizing timing parameters (p_i, e_i, d_i).

The inter-arrival time p_i is the time to the next occurrence for the trigger event that invokes the task. Those trigger events are described more in detail in Section 2.6.2. Whenever a task trigger event occurs, the corresponding task T_i releases a job $J_{i,j}$. A job $J_{i,j}$ is the j^{th} instance of task T_i and progresses the software functions that are included in the tasks in order to fulfill the logical constraints of the embedded real-time system. The execution time e_i is the amount of time units that are required for processing task T_i. The execution time e_i depends on the hardware system (processing frequency) where the task is performed. The relative deadline d_i is the temporal limit of a job T_i. The inter-arrival time p_i and the relative deadline d_i are environmental constraints and thus platform-independent. Each job should finish its execution in not more than d_i time units after its release. If this requirement is fulfilled, the deadline is *kept*, otherwise it is *violated*. A deadline violation may cause different consequences to the systems, depending on the *criticality* of the concerning task. The criticality of a task can be either hard, firm or soft, like described in Section 2.1. Task models can be distinguished as to their types of deadlines. A task set τ has *implicit deadlines* if $d_i = p_i$ for each task $T_i \in \tau$. Further, *constrained deadlines* are applied if $d_i \geq p_i$ for each task $T_i \in \tau$. A task set is said to have *arbitrary deadlines* if deadlines are neither *implicit* nor *constrained*. For tasks with arbitrary deadlines it is possible that $d_i > p_i$ so that task instances of a task T_i may overlap without deadline violations.

2.3.2 Jobs

A job $J_{i,j}$ is the unit that is executed on a core and processes the calculation of the functions that are assigned to its belonging task T_i. Whereas a task contains a relative deadline d_i as timing parameter, a job $J_{i,j}$ has an absolute deadline $D_{i,j}$ instead. The absolute deadline $D_{i,j}$ can be calculated as follows:

$$D_{i,j} = d_i + a_{i,j} \tag{2.1}$$

Where $a_{i,j}$ denotes the activation time. That is the point in time when the job is released by a task trigger. The activation time depends on the inter-arrival time p_i and the instance number j of the job $J_{i,j}$.

$$a_{i,j} = j \cdot p_i \tag{2.2}$$

Where j is the j^{th} instance of a task T_i and p_i is the inter-arrival time of task T_i. The absolute deadline $D_{i,j}$ is that point in (global) time where the job has to be finished. Otherwise, if job $J_{i,j}$ terminates after $D_{i,j}$, it would violate its deadline.

2.3.3 Runnables

The concept of runnables is derived from OSEK/VDX OS [116] and AUTOSAR standard [14]. The functionality of the whole system is divided into runnables. Runnables are a block of code that correspond to a high-level functionality, typically a function [23]. Runnables are the units that provide the runtimes to the system. Depending on their temporal requirements, runnables are mapped to real-time tasks and processed on a core whenever a job $J_{i,j}$ of a Task T_i is dispatched. The concept of runnables originates from the interface-based design approach, presented by De Alfaro and Henzinger: The system designer, composing a set of runnables, only needs to understand the interface of the runnable and not the details of how the functionality offered by the runnable is implemented. This enables to integrate a set of runnables if their input and output signals 'match' [56].

2.4 Resources

In complex embedded real-time systems, tasks are not independent from each other. That means that they have to share resources. A real-time system contains a set of resources R with n resources $(R_1, ..., R_n)$. A resource that is accessed by two different tasks at least is called a *shared resource*. In contrast, a resource that is accessed by only one task is called *private resource*. Buttazzo gives some typical examples of resources in [36], as there are data structures, sets of variables, main memory areas, files, or sets of registers of peripheral devices.

2.5 Scheduling Problem

In context of this thesis, we make use of the definition from Blazewicz et al. to formulate a description for the generic scheduling problem:

Definition 2.4. Scheduling is said to be the assignment of processors P from a set of processors P and resources R of a set of resources R to a task T of the task set τ [25] .

Garey and Johnson showed that this problem of assigning multiple tasks to multiple processors is NP-complete [74]. In general, there exist different approaches to scheduling algorithms for multiprocessors, for instance global, local, or clustered scheduling of tasks. A classification and a more detailed description of multiprocessor real-time scheduling algorithms are provided in Chapter 3.2.

2.6 Real-Time Examination

For real-time examination, several definitions are needed regarding the temporal requirements of real-time tasks. Two problems in real-time theory concern the *feasability* and *schedulability* of real-time task sets.

Definition 2.5. Feasibility: A task set τ is *feasible* according to a given system if there exists any scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadlines. [55].

In contrast to the term of *feasibility*, *schedulability* is defined as follows:

Definition 2.6. A task T_i is said to be *schedulable* with respect to a given scheduling algorithm if it is shown that all of its jobs $J_{i,j}$ meet their deadline. A task set τ is *schedulable* if it can be shown that all of its tasks T_i within τ are schedulable [55].

Further, the concept of *sustainability* was presented by [19]:

Definition 2.7. A scheduling algorithm is *sustainable* if and only if the schedulability of any task set compliant with the model implies schedulability of the same task set modified by either decreasing execution times, increasing inter-arrival times or increasing deadlines.

A *schedulability test* is a procedure that establishes whether a feasible task set is schedulable under a given scheduler. In the context of hard real-time, a task set is schedulable if each job meets its deadline [101] whereas in soft real-time systems a task set is schedulable if its *tardiness*[1] is bounded [64].

[1] A job $T_{i,j}$ is said to be *tardy* if it violates its deadline.

A schedulability test is denoted *sufficient* if all of the task sets that are schedulable according to the test are in fact schedulable. Further, a schedulability test is said to be *necessary* if all of the task sets that are unschedulable according to the test are in fact unschedulable [55].

In general, one can differ between three different types of schedulability tests, as there are *pessimistic*, *optimistic* and *exact* ones. Pessimistic schedulability tests are developed in the context of real-time system theory, where hard real-time tasks are assumed and thus any deadline violation of a task set τ is not acceptable [59]. We define a pessimistic schedulability test as follows:

Definition 2.8. A schedulability test is *pessimistic* if it surveys sufficient conditions for schedulability (a positive result guarantees that all deadlines are always met).

Sufficient but not necessary tests have a lower runtime complexity, but they are pessimistic [126]. In contrast, optimistic schedulability tests rather judge a task set as schedulable, when the method can not solve the problem more precisely [59]. Because of this fact, the term *test* is not correct. Those tests are called *schedulability examinations*.

Definition 2.9. A schedulability examination is *optimistic* if it surveys necessary conditions for schedulability (at some point there might be a deadline miss during the execution of the system).

The last category, exact schedulability tests, always calculate the exact result.

Definition 2.10. A schedulability examination is *exact* if it surveys necessary and sufficient conditions for schedulability.

Those sufficient and necessary (exact) tests are ideal, but intractable for many computational models [126].

Within the next sections, necessary elements of real-time examination are presented. First, different time events that occur to a job $J_{i,j}$ are introduced. Followed by this, definitions for real-time metrics are given, as well as different types of activation pattern for real-time tasks. Further, several methods for real-time examination are presented, namely response time analysis, real-time calculus, model checking and finally event based simulation.

2.6.1 Time Events Of Real-Time Tasks

This section treats all those time events that might occur during task execution which are necessary for real-time examination. Figure 2.1 presents an example schedule for a

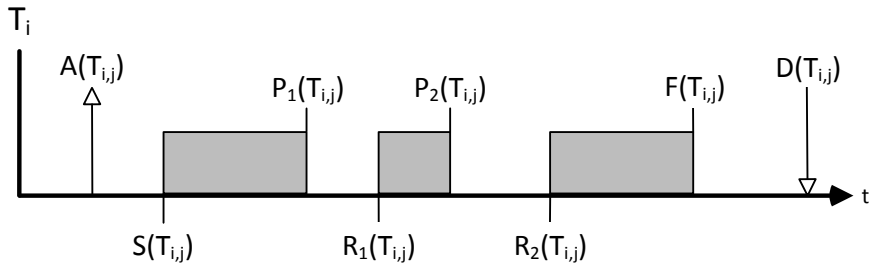

FIGURE 2.1: Time events during execution of job $J_{i,j}$.

job $J_{i,j}$ of task T_i. At the time stamp $A(J_{i,j})$, the job $J_{i,j}$ is activated according to its specified activation pattern. The execution of job $J_{i,j}$ starts at time stamp $S(J_{i,j})$. The difference between $A(J_{i,j})$ and $S(J_{i,j})$ may have different causes. First, the scheduler needs calculation time to assign the job to a processing unit. Further, job $J_{i,j}$ may be delayed by another job $J_{a,b}$, that is activated at the same time. Job $J_{i,j}$ is executing until it reaches time stamp $P_1(J_{i,j})$. At this point, job $J_{i,j}$ gets preempted from its processing unit. An example for this is if another job $J_{a,b}$ with higher priority than job $J_{i,j}$ gets activated. Followed by this, the scheduler decides to preempt job $J_{i,j}$ and lets job $J_{a,b}$ execute. When time stamp $R_1(J_{i,j})$ is reached, job $J_{i,j}$ gets resumed, which means that it continues execution. In this example, job $J_{i,j}$ gets preempted ($P_2(J_{i,j})$) and resumed ($R_2(J_{i,j})$ once again. The job finishes its execution at time stamp $F(J_{i,j})$. Because of the fact that the time stamp of finish $F(J_{i,j})$ occurs before the time stamp of the absolute deadline $D(J_{i,j})$, job $J_{i,j}$ keeps its deadline and thus fulfills its timing requirements.

2.6.2 Job Activation Patterns

The activation of a Job $J_{i,j}$ depends on the environment of the real-time system. Different types of time-bases for job activation may be required. The kind of activation pattern of a real-time system has crucial consequences to the real-time examination.

The following sections provide an overview of different time-bases that might occur in real-time systems.

2.6.2.1 Periodic Activation

The task model with periodic activation was first described by [101]. It assumes a constant inter-arrival time p_i between all activations of jobs $J_{i,j}$ of task T_i. The periodic task model contains one further timing parameter compared to the task model in Section 2.3.1: the offset o_i. The offset of a periodic task denotes the point in time from system start, when the first activation of a task T_i (job $J_{i,0}$) is released and becomes available for execution. The following job $J_{i,1}$ is released after the inter-arrival time p_i is expired. Practical examples of embedded systems with periodic task activation patterns are sensory data acquisition, control-loops, action planning or system monitoring [36].

2.6.2.2 Sporadic Activation

Sporadic task activation occurs when jobs are released as a reaction from asynchronous, external events from the environment of the real-time system. Mok describes a model for sporadic activation [110]. Sporadic tasks differ from periodic tasks in the fact that the inter-arrival time p_i denotes the *minimum*, rather than exact separation between successive jobs of the same task [20]. Examples of sporadic task activation are the arrival of bus messages from CAN or FlexRay.

2.6.2.3 Angle-Triggered Activation

Angle-triggered task activation is a special activation pattern in the field of automotive. The trigger source for releasing jobs is the crank shaft in automotive engine control systems. It is necessary to activate jobs depending on cylinder positions in order to guarantee that the optimal point in time for actions like fuel injection or ignition is kept. Angle-triggered activations can be seen as periodic activations which have their own time base (clock). Recent work that considered this activation pattern as well are [59] and [24].

2.6.3 Real-Time and Performance Metrics

In this section, metrics for real-time examination are presented. A metric can be seen as a measurement function, and we apply the definition of measurement from [87]:

Definition 2.11. 'Measurement is the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them.' [87]

In terms of software, a metric is a measurement function of some property of a piece of software or its specification.

2.6.3.1 Utilization

The utilization of a task T_i is the portion of one processor that it requires for its execution. The utilization, independent from its activation pattern, is the ratio of its parameters, the execution requirement e_i to its period pi:

$$U(T_i) = \frac{e_i}{p_i} \tag{2.3}$$

Further, the utilization of a task set is the sum of the utilizations for all tasks in τ:

$$U(\tau) = \sum_{T_i \in \tau} U(T_i) \tag{2.4}$$

2.6.3.2 Response Time

The *response time RT* of a job $J_{i,j}$ describes the time it requires to finish from the point in time of its activation.

$$RT(J_{i,j}) = F(J_{i,j}) - A(J_{i,j}) \tag{2.5}$$

2.6.3.3 Normalized Response Time

The response time RT of a job $J_{i,j}$ can be normalized according to its relative deadline:

$$NRT(J_{i,j}) = \frac{RT(J_{i,j})}{d(J_{i,j})} \tag{2.6}$$

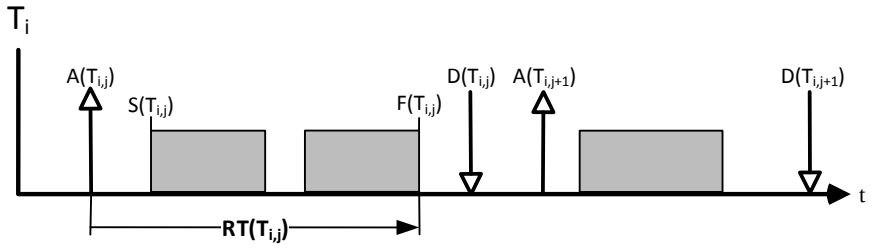

FIGURE 2.2: Job metric response time of job $J_{i,j}$, measured from activation $A(J_{i,j})$ to finalization $F(J_{i,j})$.

The *normalized response time NRT* is the fundamental metric to measure the deadline compliance of a job ($RT_{i,j} < d_i$). If the value of NRT of a job $J_{i,j}$ is smaller than or equal to 1.0, the job has kept its deadline. Otherwise, if the NRT is greater than 1.0, the deadline was violated.

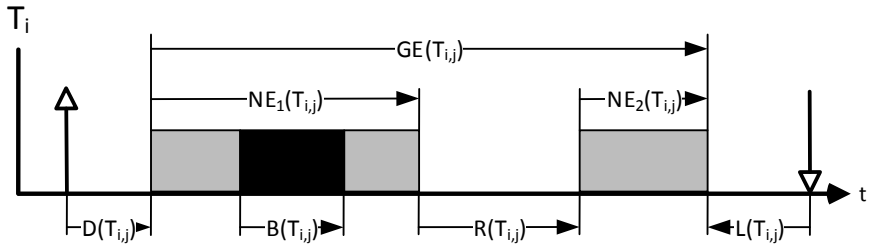

FIGURE 2.3: Depiction of different real-time metrics regarding job $J_{i,j}$.

2.6.3.4 Start Delay

The start delay denotes the time from activation $A(J_{i,j})$ to the execution start $S(J_{i,j})$ of a job $J_{i,j}$.

2.6.3.5 Netto Execution Time

The *Netto Execution Time NE* of a job $J_{i,j}$ denotes that part of the response time where $J_{i,j}$ is running on a processor. Parts where $J_{i,j}$ is waiting for the permission to access a resource while staying active on the processor is a part of NE as well.

2.6.3.6 Blocking Time

The time where an active job $J_{i,j}$ does not make progress despite it is executing on a core because it is waiting for another job $J_{a,b}$ to release a shared resource, is called *blocking time B*.

2.6.3.7 Gross Execution Time

The *gross execution time GE* denotes the time from where a job $J_{i,j}$ starts executing ($S(J_{i,j})$) to the point of time where it is terminated ($F(J_{i,j})$). It can also be calculated as the sum of NE and the ready time R, where the job $J_{i,j}$ is suspended from the core.

$$GE(J_{i,j}) = NE(T_{i,j}) + R(T_{i,j}) \tag{2.7}$$

2.6.3.8 End-to-End, Start-to-Start

For systems which require a constant inter-arrival time of periodic calculations, two further metrics can be used. The End-to-End jitter $E2E_{i,j}$ describes the difference of the finalization of two subsequent jobs ($J_{i,j}$ and $J_{i,j+1}$) of a task.

$$E2E_{i,j} = F(J_{i,j+1}) - F(J_{i,j}) \tag{2.8}$$

Analog to that, the Start-to-Start jitter $S2S_{i,j}$ describes the difference of the start of two subsequent jobs ($J_{i,j}$ and $J_{i,j+1}$) of a task.

$$S2S_{i,j} = S(J_{i,j+1}) - S(J_{i,j}) \tag{2.9}$$

The goal of the systems with the requirement described above is to keep these jitters as small as possible. A more detailed description of E2E and S2S can be found in [59].

Chapter 3

Background and Related Work

This chapter provides the prior work on which the contributions of this thesis are based. Methods for real-time analysis are discussed as well as related work in real-time multiprocessor scheduling and real-time synchronization mechanisms.

3.1 Methods For Real-Time Analysis

The focus of this thesis lies in the second requirement of Definition 2.1, that the correct system behavior of a real-time system depends on its temporal requirements (deadlines). Thus, methods for real-time analysis are necessary in order to assess real-time systems regarding their temporal requirements. Within the next sections, different methods are presented that can be applied for real-time analysis. These different methods are the Response Time Analysis (RTA), The Real-Time Calculus (RTC), the model checking approach and finally Discrete Event Simulation (DES).

3.1.1 Response Time Analysis

The response time analysis RTA is an exact analytical schedulability test. The main idea of RTA is to determine the worst case response time WCRT by a fix point iteration, with the goal to find the so called *critical instance* [85]. Equation 3.1 shows the basic iterative formula to calculate the response time R_i of a periodic task T_i. For all tasks with higher priorities $hp(T_i)$, it is calculated how many activations of the task T_j with a

minimum inter-arrival time p_j enter during the response time R_i^n of the current iteration n. The determined activation number gets multiplied by the execution time e_j. Finally, the execution time e_i of task T_i is added. The result is used to determine the response time in the next iteration. The iteration aborts if R_i^n is equal to R_i^{n+1}.

$$R_i^{n+1} = e_i + \sum_{j \in hp(T_i)} \left\lceil \frac{R_i^n}{p_j} \right\rceil \cdot e_j \tag{3.1}$$

Two bounds of response time can be determined with the help of this approach: The worst-case bound and the best-case bound. It is guaranteed that all observable response times will fall between those two bounds [82]. The RTA developed in [85] can be applied for periodic task sets and task-fix priority scheduling. RTA for periodic task sets with task-fix priorities and arbitrary deadlines was presented by [80]. Extensions proposed by [12] contain features like release jitters or conditioned deadlines. For uniprocessors it has been shown in [19] that RTA is sustainable regarding all task parameters, even if non-preemptive resources are shared between different tasks. Equation 3.2 shows the extended formular for RTA where an additional blocking time is considered. This blocking time occurs if resource sharing is applied and a synchronization mechanism is used (e. g. semaphores).

$$R_i^{n+1} = e_i + B_i \sum_{j \in hp(T_i)} \left\lceil \frac{R_i^n}{p_j} \right\rceil \cdot e_j \tag{3.2}$$

Where B_i denotes the blocking time for task T_i in case that it has to wait until a required resource is available. Nevertheless, to the best of our knowledge, no RTA algorithm exists that is able to calculate the bounds of task sets for multiprocessor systems with different types of activation events and nested shared resources.

3.1.2 Real Time Calculus

Originally, the real-time calculus RTC [43] is a real-time examination for distributed embedded systems which is based on the widely used Network Calculus [97].

It uses a count based abstraction to determine upper and lower arrival curves of response times. Those curves define the maximum and minimum number of events as a function of the time interval $\Delta\ t$. The RTC can be applied to different types of task sets with either periodic or sporadic activation etc. [117]. An extension of the RTC was proposed

by [99] which makes it possible to analyze task sets scheduled by global, job-fix priority scheduling algorithms. Further, [123] presented a variant of RTC combined with a probabilistic approach which abstracts the functional and non-functional requirements of real-time components in a probabilistic model.

3.1.3 Model Checking

Model checking is another possible approach for real-time system examination. Clarke and Schlingloff [48] define the term of model checking as follows:

Definition 3.1. 'Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems.'

Model checking algorithms that were developed for real-time examination mostly are realized by applying timed automata (TA) [4]. TA are extensions of finite automata with a finite number of control states, clocks with continuous time and conditions regarding time (as invariant or access barrier). An advantage of TA is that periodic task sets can be analyzed as well as non-periodic and sporadic task sets [79]. Further, TA can be combined to networks.

[108] presented a TA model checking approach for multicore embedded systems, where systems with partitioned, static and dynamic priority scheduling algorithms can be checked. This tool enables schedulability analysis as well as verifications of memory usage and power consumption. The model-checking technique from [79] provides an exact schedulability test for both, partitioned and global scheduling for task sets with static priorities. Systems with hierarchical scheduling can be analyzed by the model-checking approach of [107] which is further able to consider resource sharing. A model-checking analysis specifically for automotive powertrain systems was provided by [75], which considers partitioned scheduling of periodic task-sets with task-fix priorities and resource sharing.

3.1.4 Discrete Event Simulation

Discrete event simulation DES is an optimistic schedulability examination that uses a temporal behavior model to estimate worst case response time. As described in [59], DES has two characterizing properties. First, the model parameter values are discretized

to describe the functional system behavior. Secondly, the time when values change (event) is discretized as well to describe the temporal system behavior. In figure 3.1, an example for the discretization of model parameter values is given, while figure 3.2 shows the discretization of time, as it is presented in [60]. This approach is close to embedded systems in that manner that those systems work mainly with digital components with parameter granularity bit and timing granularity frequency. The model for DES consists

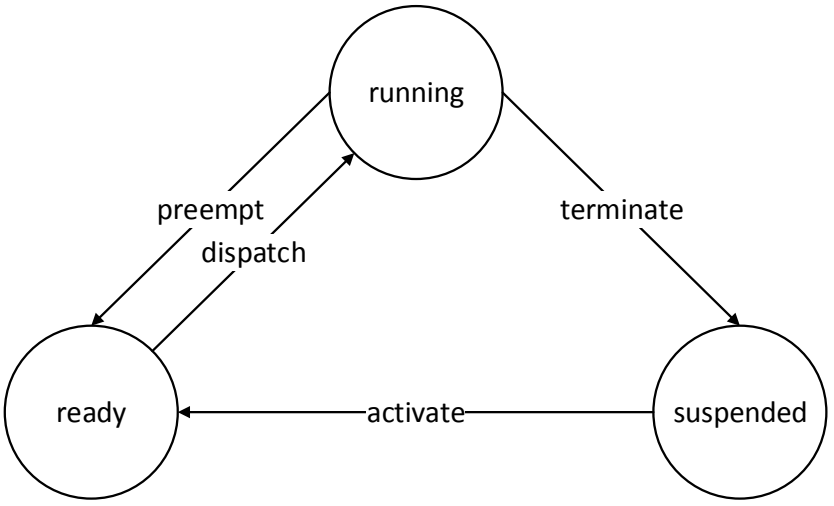

FIGURE 3.1: State model of the component *task* with discrete states and conditions for state transition. State transitions occur by interactions with other state machines or other components with the same state model and may be accompanied by delays.

of a number of components, each of them is described by a state machine. The state model is additionally enriched with delays so that temporal behavior can be described properly. An example for such a delay is the runtime of a function that is taken for instance from a hardware trace. State transitions occur in interaction between the different components. By means of this technique, a behavioral model of hardware, operating system and software can be created. For a more detailed description of DES, interested readers are referred to [59].

For instance, *MAST* [81] is such an approach which allows to model and analyze (feasibility tests and sensitivity analyses) distributed real-time systems. Recently, different simulation approaches [128, 122, 136, 44, 46] were presented that provide schedulability tests for different scheduling algorithms. The first simulator that deals with execution dependencies was presented by [106], where randomly varied execution times are used to estimate an approximation (by extreme value theory [22]) of the maximum response time by carrying out multiple simulation runs.

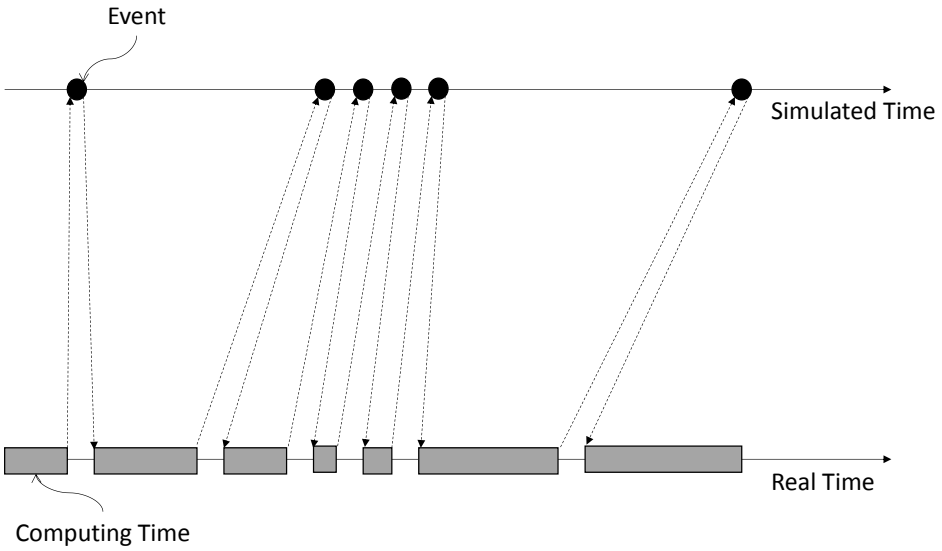

FIGURE 3.2: Depiction of simulated time versus real (computing) time as it occurs in discrete event based simulations [60]. The simulation steps from event to event, dependent on the state transition, while the real time makes progress according to the calculation time.

Deubzer et al. presented an event based simulator [60] that allows not only to analyze different scheduling algorithms but also synchronization mechanisms for shared resources and monitoring of event-chains. It allows randomly varied execution times and different activation patterns (see Section 2.6.2) as well.

3.2 Multiprocessor Real-Time Scheduling

The scheduling problem was already formulated in Chapter 2.5. As the focus of this thesis lies in shared memory multiprocessors (multicore systems), we skip the description of the well-known topic of single-core scheduling and continue with multiprocessor scheduling. First, some definitions regarding scheduling algorithms are presented. Followed by this, a classification of multiprocessor scheduling algorithms is given and explained. Finally, an overview of related work regarding the different classes of multiprocessor scheduling algorithms is shown. Note, that all descriptions regarding schedulability in these sections are made with the assumption that tasks are independent (without blocking times).

3.2.1 Definitions

Whenever a task is activated, a job is generated and inserted into a list (or job-queue) of activated tasks. The job-queue contains all jobs that have to be assigned to a core in order to get executed. The applied *scheduling algorithm* now builds a *schedule* to assign the activated jobs to a core dependent on a given *scheduling policy*. A real-time scheduling policy has to find a schedule in a way so that all jobs meet their deadlines. Davis and Burns give a definition of optimal real-time scheduling algorithms.

Definition 3.2. 'A scheduling algorithm is said to be optimal with respect to a system and a task model if it can schedule all of the task sets that comply with the task model and are feasible on the system' [55].

After the scheduler built the task schedule, the highest rated job is dispatched and thus executed on the selected core. The scheduler has to reschedule each time a job is either activated or terminates, or a schedule event is triggered (for example if a *schedule-point* is reached). Further, if a priority-driven scheduler is applied, it has to reschedule if the priority of at least one job changes. A *dispatcher* manages the execution of a schedule and performs necessary steps like context switching.

3.2.2 Classification Of Multiprocessor Real-Time Scheduling Algorithms

Deubzer presented a classification of multiprocessor real-time scheduling algorithm in [59]. Based on this work, we present a slightly different version which is shown in Table 3.1.

TABLE 3.1: Classification of multiprocessor real-time scheduling algorithms.

	I	II	III	IV
Allocation	Partitioned	Semi-Partitioned	Clustered	Global
Disruption	Non-Preemptive	Cooperative	Preemptive	
Migration	No	Job-	Section-	Fully
Prioritization	Task-fix	Job-fix	Section-fix	Dynamic
Processing	Work-Conserving	Non-Work-Conserving		

The difference to the classification in [59] is, that we added the term of semi-partitioned allocation. In the following, an explanation of the different categories of scheduling algorithms is given.

3.2.2.1 Allocation

The category *allocation* denotes how active jobs can be assigned by the scheduler to the different cores. Partitioned allocation (A-I) means that every core of the multiprocessor has its own scheduler and tasks and thus their generated jobs are assigned statically to one specific scheduler.

Semi-partitioned allocation (A-II) can be seen as a hybrid version of A-I and A-IV. Generally, all tasks are assigned to a fix core, but some tasks get the permission to migrate on another core.

When using clustered allocation (A-III), a subset of cores is managed by a scheduler, and tasks are assigned statically to one scheduler. The difference to local allocation is, that activated jobs can be assigned to different cores within a cluster.

Finally, global allocation (A-IV) means that there exists one single scheduler that assigns activated jobs to a core according to its policy. Tasks are not statically assigned to a core or a cluster of cores in this approach.

3.2.2.2 Disruption

Disruption defines points where executing jobs may be preempted by another (pending) job that has a higher rank according to the schedule calculated by the scheduling algorithm. Deubzer distinguishes between two different types of disruption, *preemption* and *interruption* [59]. Preemption is costly because disruption is caused by another job and the complete task context has to be saved. In contrast, *interruption* occurs by an ISR where only a subpart of the context has to be saved (due to limited functionality of ISRs) and thus is less costly.

The first manner of disruption is *non-preemptive* (D-I). Once a job $J_{i,j}$ is running, it can not be preempted by another job $J_{a,b}$ until $J_{i,j}$ terminates, even if $J_{a,b}$ has a higher rank.

At *Cooperative* (D-II) scheduling, a running job $J_{i,j}$ can only be preempted when a certain, defined preemption point is reached. Mainly, two different approaches exist in case of cooperative scheduling. With fixed preemption points [35] each task is divided statically into a set of non-preemptive sections, it can only be preempted at the defined preemption points. In contrast, there also exists the concept of floating non-preemptive regions [139] where a running task can switch into a (bounded) non-preemptive execution phase. Preemptions through other, higher priority tasks, can be delayed to a later point in time.

Preemptive (D-III) scheduling means that a running job can be preempted at every point in time during its execution.

3.2.2.3 Migration

Migration can only be applied when either global or clustered allocation is used. The following types of migration rules are restrictions on how jobs may be allowed to execute on different cores.

No migration (M-I) means that each job $J_{i,j}$ of a task T_i has to be executed on the same core which is the case at partitioned scheduling. Such tasks are said to have *core affinities*.

Bounded migration (M-II) means that a job $J_{i,j}$ has to be executed completely on that core where it is assigned to by the scheduler, but the next instance of the same task, job

$J_{i,j+1}$, may be executed on a different core.

Task section migration (M-III) means that a job $J_{i,j}$ is allowed to migrate to another core only at a defined preemption point.

Finally, *full migration* (M-IV) means that a job $J_{i,j}$ is allowed to migrate to another core at any point in time.

3.2.2.4 Prioritization

Prioritization defines the possible granularity where priorities of a job can be changed during execution.

When *Task-fix prioritization* (P-I) is applied, the priority of a task T_i and all its generated jobs is defined offline and remains constant during the whole execution of the system.

Job-fix prioritization (P-II) means that the priority of a job $J_{i,j}$ is calculated at the activation of a job and does not change until $J_{i,j}$ is terminated. The successor of $J_{i,j}$, $J_{i,j+1}$ may have a different priority.

When using *Section-fix prioritization* (P-III), the priority of a job $J_{i,j}$ may change at a schedule point.

Dynamic prioritization (P-IV) means that the priority of a job $J_{i,j}$ can change at any point in time.

3.2.2.5 Processing

Processing defines how a scheduler allocates jobs that are ready for execution. *Work-conserving* (W-I) means that an activated job $J_{i,j}$ is assigned to a core as soon as there is a core available.

In contrast, *non-work-conserving* (W-II) means that a core might be left in idle mode, even if there exists at least one activated job $J_{i,j}$.

3.2.3 Partitioned Multiprocessor Scheduling

Like already described in 3.2.2, tasks are allocated statically to the corresponding cores in the case of partitioned scheduling. This approach was investigated first by [65] in the context of real-time systems.

Figure 3.3 shows an example where partitioned scheduling is applied to 9 tasks and 3 processing units (cores). Task 1 - Task 3 are partitioned to Core 1, Task 4 - Task 6 are allocated to Core 2 and the jobs of Task 7 - Task 9 execute only on Core 3. Partitioned

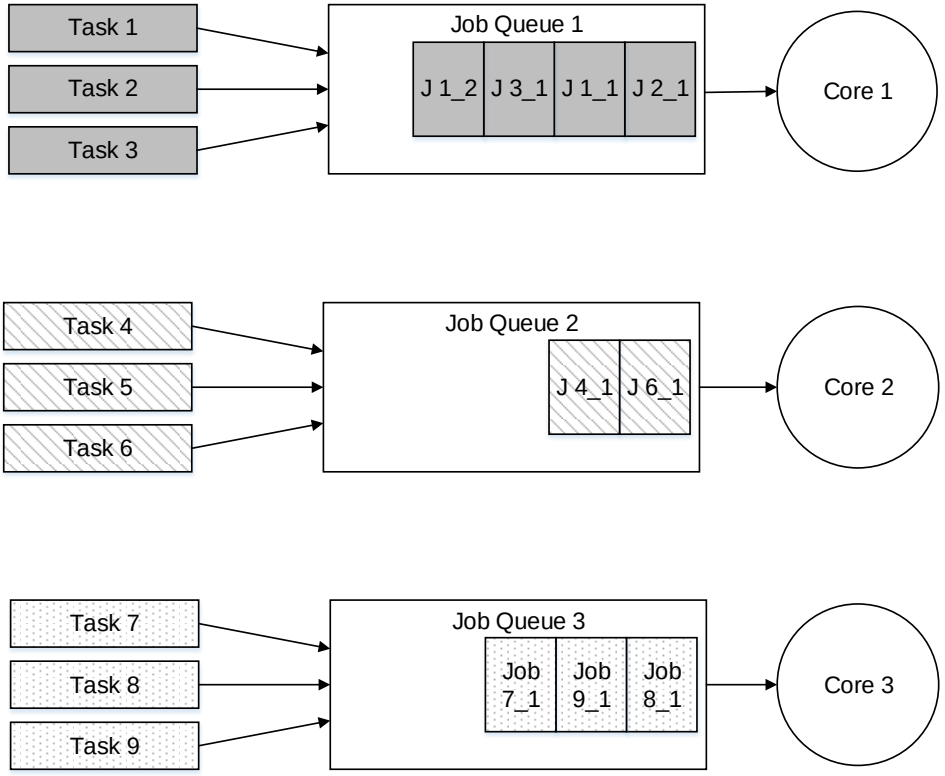

FIGURE 3.3: Example of a partitioned task-to-core allocation where 9 tasks are assigned to 3 available cores. When a job $J_{i,j}$ of a task T_i gets activated, it is added to the job queue of the corresponding core. Afterwards, the scheduler of this core decides which job within the job queue is selected for execution.

scheduling is accompanied by the following advantages compared to global scheduling. First, there are no migration costs to take into account because a task is always processed on the same core. Further, the usage of per-processor job-queues might cause a lower scheduling overhead (compared to one single job-queue for all tasks) if a high number

of tasks has to be managed. Another advantage is that each partition can be scheduled with the algorithms for uniprocessor scheduling. Finally, real-time analysis techniques can also be applied from uniprocessors if resource sharing is not considered or tasks are independent across the different partitions. However, up to now no optimal scheduling algorithm exists for partitioned allocation.

In the next sections we first describe how to partition (assign tasks to a certain core) a task set in a manner that no processor is overloaded. Further, it is considered what kind of scheduling policy can be used to schedule the partitions in a way that feasible task sets are schedulable.

3.2.3.1 Task-Partitioning

The proceeding of task partitioning is a kind of bin-packing problem, which is known to be NP-hard in the strong sense [74]. The classic bin-packing problem can be formulated as follows:

Given a fix bin capacity V and a set of boxes x_1,...,x_n with sizes a_1,...,a_n to pack. The target is to find an assignment for each box into a bin in a way that the V is not exceeded and the (integer) number of bins B is minimized.

Adapted to partitioned scheduling, processors correspond to bins and boxes correspond to tasks. The available capacity of a core $C(P_n)$ indicates the bin capacity V and the size of a box a_x indicates the task utilization $U(T_x)$.

However, the difference to the classic bin-packing problem here is that the number of bins is given by the amount of available cores. If a higher number of bins was required to find a bin-packing solution in terms of assigning tasks to cores, the task set would be simply not schedulable.

To come to a solution for the task assignment problem, one have to define the order of selecting the tasks and further the order of selecting the cores. For selecting the tasks, the decreasing order (in terms of utilization) is preferable against increasing order, as large items are more difficult to pack than smaller items and the probability to fit into the remaining capacity of a partially used processor is higher for smaller items. López et al. showed that order tasks by decreasing utilization achieve higher utilization bounds [103]. Because of this, we focus on the description of bin-packing in decreasing order

only. In the following parts an overview is given whether the processor for the actual bin may be selected.

a) First-Fit Decreasing The first-fit decreasing bin-packing takes the first item (task) from the sorted list and assigns it to the first core (cores are sorted by their index) with a sufficient capacity left.

b) Best-Fit Decreasing The best-fit decreasing order considers all available cores and select that one which will have the minimal remaining capacity after placing the task. If no such bin exists, an empty one is chosen (sorted by the index).

c) Worst-Fit Decreasing In contrast to best-fit decreasing, that core is selected which will have the maximum remaining capacity after placing the task.

d) Next-Fit Decreasing The next-fit bin-packing is similar to the first-fit decreasing except for the fact that the order of cores does not start at the core with the lowest index, but rather at the next index corresponding to the last selected core.

3.2.3.2 Task-Scheduling

Different well studied scheduling policies exist for the approach of local allocation of tasks. In the next sections, a short overview of the most discussed algorithms is given. We categorize algorithms for different types of prioritization according to the classification in Section 3.2.2. As a more detailed discussion of these algorithms is out of scope of this work, interested readers may be referred to [55].

a) Task-Fix Priority Scheduling (P-I) Task-fix priority scheduling is not optimal for uniprocessor scheduling and thus it is neither optimal in case of partitioned multiprocessor scheduling. The maximum utilization bound for task sets τ executed on m cores by applying (partitioned) task-fix priority scheduling is $U_{sum}(\tau) \leq (m+1)/\left(1+2^{1/(m+1)}\right)$. [1] This was proven by [114].

[1] The utilization bound converges to $U_{sum}(\tau) \leq \frac{m+1}{2}$ for a high number of processors (m).

In more recent work, [10] showed that the utilization bound for implicit deadline task sets with hard real-time requirements is $U_{sum}(\tau) \leq \frac{m}{2}$.

Task-fix priority scheduling algorithms can be distinguished according to their priority assignment. The following part gives a short overview of the algorithms *Rate Monotonic* (RM), *Deadline Monotonic* (DM) and Audsley's priority assignment.

i) Rate-Monotonic (RM) presented by [101] is an assignment policy for periodic task sets. Priorities are assigned according to the inverse of the task period.[2] It has been shown that RM is the best priority assignment for (periodic and sporadic) task sets with implicit deadlines at uniprocessors [101].

ii) Deadline-Monotonic (DM) assigns priorities according to the inverse of the relative deadline of a task. It was developed by [100]. They have also shown that DM outperforms RM priority assignment for task sets with constraint deadlines.

iii) Optimal Priority Assignment (OPA) from [13] is known to be the best algorithm (at task-fix prioritization) for asynchronous periodic task sets or task sets with arbitrary deadlines. OPA recursively determines a priority assignment by considering at most O(n^2) distinct priority assignments.

b) Job-Fix Priority Scheduling (P-II) For job-fix priority scheduling it is known that *Earliest Deadline First* (EDF) from [58] is an optimal algorithm for uniprocessor scheduling when sporadic task sets (independent from the deadline constraints) are applied. However, Dertouzos and Mok have shown that optimal uniprocessor algorithms are no longer optimal for partitioned scheduling [57]. López et al. have proven that task sets with implicit deadlines are schedulable on m processors with hard real-time constraints under partitioned EDF scheduling if either first-fit, best-fit or worst-fit decreasing bin-packing is applied and further if the utilization of the task set $U_{sum}(\tau) \leq \frac{m+1}{2}$ in [104].

Priorities within the EDF scheduling algorithm are assigned to jobs according to their absolute deadline (in increasing order). That means that the job with the earliest absolute deadline gets the highest priority.

[2] The task with the shortest period gets the highest priority.

A disadvantage regarding the overhead of the algorithm is that the absolute deadline has to be calculated at every activation of a job as there is no static (fix) priority. On the other hand, there are two advantages that compensate that drawback. First, EDF is robust against permanent overload, which has been proven by [42] and second, when EDF is applied, the number of context switches and there is a lower number of preemptions when task-fix priority scheduling is performed.

Experiments of Buttazzo revealed that the average number of preemptions decreases for high loads under EDF, whereas for RM it increases almost linearly [37].

c) Dynamic Priority Scheduling (P-IV) As EDF is optimal (for uniprocessor scheduling), there is no need to use a fully dynamic priority assignment. It further has the disadvantage that the priorities have to be determined not only once for a job, but every time the scheduler is called. However, Mok developed the *Least Laxity First* (LLF) algorithm [110]. *Laxity* is the largest time for which the start of execution of a job can be delayed without violating its deadline.

$$L_{i,j} = D_{i,j} - a_{i,j} - e_{i,j} \tag{3.3}$$

LLF assigns the highest priority to the job with the lowest laxity. The laxity of executing jobs remains constant, whereas the laxity of activated jobs (in state ready) decreases continuously. This can lead to multiple preemptions of tasks without dispatching a new task, which causes many context switches and thus overhead. Additionally, there exists the problem that the calculation of laxity is inexact, because the worst case is always assumed. A modified LLF algorithm presented by [115] reduces the number of context switches by allowing laxity inversions as far as deadlines are kept.

3.2.4 Semi-Partitioned Multiprocessor Scheduling

Semi-Partitioned Multiprocessor Scheduling can be seen as a special variant of partitioned allocation, where tasks are assigned statically to a certain processor. However, a selection of some tasks is allowed to migrate under some assumptions. EKG (**E**DF with task splitting and **k** processors in a **g**roup.) from [9] is an approach for periodic task sets with implicit deadlines. Some tasks are split into two components that are scheduled at different times on different processors. EKGs utilization bound depends

on the parameter k. It is used to divide tasks into groups of heavy and light tasks. The maximum utilization of 100% can be reached for k = m (at k=2 the maximum utilization is still 66%.) [9].

Kato and Yamasaki presented EDDP (Earliest Deadline Deferrable Portion) [89] which is based on EDF and splits at most m - 1 tasks across two processors. While executing split tasks the two parts are prevented from executing simultaneously. They also showed in [89] that the utilization bound of EDDP is 65% for periodic task sets with implicit deadlines. Furthermore, Lakshmanan et al. and Kato both present semi-partitioned scheduling algorithms with task-fix priorities, based on DM scheduling. PDMS_HPTS (Partitioned Deadline Monotonic Scheduling - Highest Priority Task of a Partition is allowed to Split) [94] is only made for sporadic task sets with implicit or constrained deadlines. DM-PM (Deadline Monotonic with Priority Migration) [88] is as well made for sporadic task sets, where tasks would be allowed to migrate if they do not fit any single processor. Further, migrating tasks are assigned highest priority.

RUN (**R**eduction to **Uni**processor), a scheduling algorithm from Regnier et al. [121], is claimed to be optimal for periodic preemptive tasks with implicit deadlines. It uses abstractions to build a reduction tree (offline data structure). A dual task set is created, where tasks have the same deadlines but complementary workloads compared to the original tasks. These dual tasks represent the idle times of the processors. Followed by this the problem can be reduced iteratively to a uniprocessor scheduling problem. EDF is then applied which is optimal for uniprocessors (see Section 3.2.3.2). However, RUN seems to be difficult to be implemented in practice because additional operating system support is required. An example implementation can be found in [49]. A further disadvantage is that RUN is not able to schedule sporadic tasks which makes it non-practicable for many practical embedded real-time systems. An advantage over the later described optimal global multiprocessor scheduling policies is that RUN produces less context switches and preemptions (see case studies in [46], [49]) which leads to lower scheduling overheads.

3.2.5 Global Multiprocessor Scheduling

In contrast to partitioned scheduling, all tasks are assigned to one single scheduler in the case of global scheduling. The scheduler manages all cores and therefore every task

might be executed on every core. Figure 3.4 gives an example for global scheduling. All tasks of the task set τ (Task 1 - Task 9) are assigned to one global scheduler that distributes the activated jobs of the tasks onto the cores (Core 1 - Core 3) according to a specific scheduling policy.

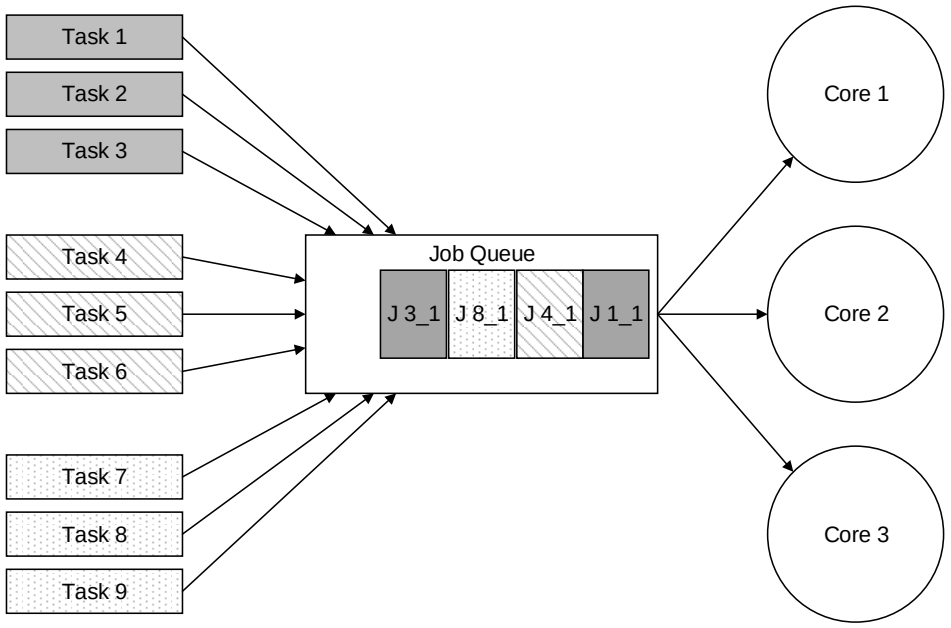

FIGURE 3.4: Example of global allocation where 9 tasks are assigned to 3 available cores. All activated jobs are added to a single job queue. The scheduler decides which job is executed on which core.

In general, global scheduling in comparison to partitioned scheduling has the advantage that there is no need to run partitioning or load balancing algorithms. Further, [8] have shown that fewer context switches occur at global scheduling because preemptions only may take place if no processor of the system is idle at this moment. Nevertheless, [69] revealed that no optimal global scheduling algorithm exists for sporadic task sets with arbitrary and constrained deadlines. However, [41] pointed out that the only possible way of achieving an optimal scheduling algorithm for periodic task sets is a dynamic priority assignment.

The next sections present a selection and a short discussion about either task-fix, job-fix or dynamic priority assignments for global scheduling.

3.2.5.1 Task-Fix and Job-Fix Priority Global Scheduling

In case of global scheduling it has been shown for periodic task sets with implicit deadlines that the utilization bound of the task-fix priority algorithm RM and the job-fix priority algorithm EDF is $U_{sum}(\tau) \leq 1+\varepsilon$ for arbitrary small ε [65]. This is called *Dhall effect*. Dhall and Liu further built a task set which shows an example for this effect.

For job-fix priority scheduling, [11] proved that the utilization bound for periodic task sets with implicit deadlines is $U_{sum}(\tau) \leq \frac{m+1}{2}$.

For sporadic task sets with implicit deadlines, a sufficient schedulability condition is $U_{sum}(\tau) \leq m - (m-1)U_{max}(\tau)$ [78]. Baruah presents further schedulability tests for different sporadic task sets under global EDF (G-EDF) scheduling in [18].

Some variants of EDF algorithms have been developed, we introduce two of them in the following. The scheduling algorithm EDF-US, proposed by [129], is an extension of G-EDF where deadlines are assigned according EDF rules if the utilization of a task T_i, $U_i \leq \frac{m}{2m-1}$. However, if $U_i > \frac{m}{2m-1}$, then T_i's jobs are assigned highest priority.
A further extension of global EDF scheduling, the *global fair lateness* (G-FL) algorithm was developed by [67]. G-FL is a job-fix priority algorithm as well where priorities are assigned according the following term:

$$\forall T_i \in \tau,\ Y_i = D_i - \frac{m-1}{m}\, C_i \tag{3.4}$$

Where Y_i is the actual priority of a task T_i and C_i denotes the worst case execution time WCET. That means that not only the absolute deadline D of a task is taken into account, but also a part of the WCET. This results in the fact that G-FL minimizes the lateness bounds of a given task set (compared to G-EDF), which is proved by [67].

3.2.5.2 Fully Dynamic Priority Global Scheduling

Like described before in Section 3.2.5, the only possible way to build optimal schedulers for global allocation is dynamic priority assignment. Up to now, two groups of scheduling algorithms are known to be optimal. These are the family of Proportionate Fair (Pfair) [17] and the group of Least Local Laxity First (LLREF) [47] algorithms. Both are based on the concept of fluid scheduling, an example for that is shown in Figure 3.5. In the

fluid scheduling model, tasks execute at a constant rate which is represented by a broken line in Figure 3.5.

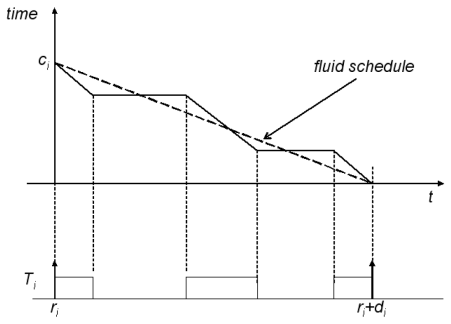

FIGURE 3.5: Example of a fluid schedule versus a practical schedule [47].

In the following we discuss these concepts and scheduling algorithms that are based on this concept.

a) Pfair Scheduling The concept of Pfair scheduling is based on a quantum-based model (a deviation of the fluid scheduling model) in which tasks make progress proportionate to their utilization (also known as *weight*). This scheduling model and the first scheduling algorithm were presented first by [17].

The Pfair scheduling model requires that task periods p_i and task execution times e_i have to be a multiple of the quantum size Q. Further, events can enter at time quanta $t = \{x \cdot Q\}$ for $x \in \mathbb{N}_0$ [59]. The received processing time *received(T_i, 0, t)* denotes the sum of time a task was executing already. The difference between the received processing time *received(T_i, 0, t)* and the fluid schedule *fluid(T_i, 0, t)* at a certain point in time is called *lag*:

$$lag(T_i, t) = fluid(T_i, 0, t) - received(T_i, 0, t) \tag{3.5}$$

A Pfair schedule is defined by Baruah et al. [17] as follows:

Definition 3.3. A schedule S of a task set τ is Pfair if and only if:

$$-1 < lag(T_i, t) < 1 \quad \forall\, T_i \in \tau \tag{3.6}$$

This definition can be explained as follows: In a Pfair schedule the cumulative allocation of a task T_i deviates by strictly less than one quantum from a fluid processor sharing

schedule at any point in time [31].

Tasks T_i in a quantum based task set consist of up to n_i time quanta which can be called *subtasks* [6] as well. The k^{th} subtask of a job $J_{i,j}$ is defined to be $J_{i,j,k}$. Each subtask has to be scheduled within a so called *Pfair window*. A Pfair window starts at its pseudo-release time $r_{i,j,k}$ and ends at its pseudo-deadline $d_{i,j,k}$. The pseudo-release time is:

$$r_{i,j,k} = a_{i,j} + \left\lfloor \frac{k-1}{U(T_i)} \right\rfloor \tag{3.7}$$

While the pseudo-deadline is defined as:

$$d_{i,j,k} = a_{i,j} + \left\lceil \frac{k}{U(T_i)} \right\rceil \tag{3.8}$$

Most of the Pfair scheduling algorithms (except Boundary Fair) make use of a tie-breaking rule called overlapping bit $b(J_{i,j,k})$ which is also known as *boundary bit* [63]. The overlapping bit has the value one if the pseudo-deadline $d_{i,j,k}$ of a subtask $J_{i,j,k}$ is later than the pseudo-release time $r_{i,j,k}$ of the successor's ($J_{i,j,k+1}$) Pfair window, otherwise it has the value zero.

$$b(J_{i,j,k}) = \left\lceil \frac{k}{U(T_i)} \right\rceil - \left\lfloor \frac{k}{U(T_i)} \right\rfloor \tag{3.9}$$

After discussing the properties of Pfair schedules, optimal scheduling algorithms that are able to handle Pfair schedules are concerned below. They all allocate subtasks according the Earliest Pseudo-Deadline First (EPDF) policy.

i) Pfair-PF The Pfair-PF algorithm has been presented (and shown to be optimal for periodic task sets with implicit deadlines) by [17]. It assigns priorities as follows: When a schedule decision takes place, when subtask $J_{i,j,k}$ and subtask $J_{a,b,l}$ are ready for execution, $J_{i,j,k}$ gets a higher priority than $J_{a,b,l}$ ($J_{i,j,k} \succ J_{a,b,l}$) if one of the following is true:

When two different subtasks that are activated have different values of pseudo-deadline:

a) $d(J_{i,j,k}) < d(J_{a,b,l})$

When two different subtasks that are activated have the same value of pseudo-deadline but differ in their overlapping bits, the higher priority is assigned to that task where the value of overlapping bit is one:

b) $d(J_{i,j,k}) = d(J_{a,b,l})$ *and* $b(J_{i,j,k}) > b(J_{a,b,l})$

If two different subtasks that are activated have the same value of pseudo-deadline and overlapping bit:

c) $d(J_{i,j,k}) = d(J_{a,b,l})$ *and* $b(J_{i,j,k}) = b(J_{a,b,l})$ *and* $J_{i,j,k} \succ J_{a,b,l}$

At each time quantum, the m tasks with the highest priorities are selected to execute for one time quantum. After that quantum, all tasks are suspended and priorities are calculated again. This behavior of the Pfair-PF algorithm causes high overheads regarding priority calculation and context switches.

ii) Pfair-PD The scheduling algorithm Pfair-PD was developed by [21]. It is called PD because it uses pseudo-deadlines for its subtasks. It contains a complex priority rule by dividing subtasks into seven different subtask classes. We focus on the algorithm Pfair-PD^2 at this point, which is an optimization of Pfair-PD.

iii) Pfair-PD^2 [6] provided a simplified variant of Pfair-PD which is called *Pfair-PD^2*. Pfair-PD^2 schedules active subtasks in order of non-increasing pseudo-deadlines. It chooses subtasks in a way that future scheduling decisions are as least constrained as possible [31].

Rule *c)* of the Pfair-PF is replaced as follows: If subtasks have an equal value of pseudo-deadline and the values of overlapping bits are zero for all subtasks, the tie in pseudo-deadlines can be broken in arbitrary order. Moreover, if subtasks have equal values of pseudo-deadline and the values of overlapping bits are non-zero, the successors of subtasks are constrained when the actual subtask gets delayed. PD^2 prefers such subtasks that would cause a longer cascade of delays. The length of such a cascade of the subtask $\mathrm{J}_{i,j,k}$ is given by the group deadline $\mathrm{D}(\mathrm{J}_{i,j,k})$ and can be calculated for non-light tasks $(0.5 \geq U(T_i) < 1)$:

$$D(J_{i,j,k}) = \left\lceil \frac{\left\lceil \left\lceil \frac{k}{U_i} U_i \right\rceil \cdot (1 - U_i) \right\rceil}{1 - U_i} \right\rceil \tag{3.10}$$

Whereas for light tasks $(0 < U(T_i) < 0.5)$ the group deadline $\mathrm{D}(\mathrm{J}_{i,j,k})$ is defined as:

$$D(J_{i,j,k}) = 0 \tag{3.11}$$

Now rule *c)* of assigning priorities in PD^2 can be formulated as:

c) $d(J_{i,j,k}) = d(J_{a,b,l})$ *and* $b(J_{i,j,k}) = b(J_{a,b,l})$ *and* $D(J_{i,j,k}) < D(J_{a,b,l})$

Pfair-PD^2 has been proven to be optimal for task sets in a multiprocessor with m cores if the following condition is true:

$$\sum_{i=1}^{n} U(T_i) \leq m \tag{3.12}$$

iv) Early Release Extension The pseudo-activation of subtasks for Pfair schedulers leads to the fact that a subtask $J_{i,j,k+1}$ is only allowed to be assigned to a core if it is pseudo-activated (time has reached pseudo-release time). This makes Pfair schedulers to non-work-conserving algorithms.

As work-conserving algorithms are generally more robust, the so called Early-Release Fair (*ERfair*) scheduling algorithm have been presented by [5]. ERfair is a work-conserving algorithm where subtasks do not have a pseudo-activation. That means that if a subtask $J_{i,j,k}$ terminates before pseudo-release time $r_{i,j,k+1}$ (if there is one) of the next subtask $J_{i,j,k+1}$ is reached, this subtask would be allowed to be processed immediately.

v) Boundary Fair Boundary Fair (BF) from [140] was developed to minimize the number of context switches which is one of the main drawbacks of Pfair based schedulers. BF makes schedule decisions only at inter-arrival time boundaries and thus allows violations of Equation 3.6 during these boundaries (A task can miss its deadline only at inter-arrival time boundaries). This algorithm is limited to periodic task sets because the task activations have to be known, so that it can be determined how many task quanta of all tasks in a task set have to be assigned to the available cores from schedule time until the next inter-arrival time boundary. After that, task quanta are placed in a manner that the number of context switches is minimal.

vi) Partly Pfair Algorithms Deubzer et al. proposed two scheduling algorithms that were constructed with the focus on automotive powertrain systems. They weakened the Pfair scheduling model and called it *Partly Pfair*. The first algorithm is called *Partly Pfair-PD*^{2} [61]. It is based on Pfair-PD^2, however it does not fulfill the proportionate fair bounds from Equation 3.6. The main properties of Partly Pfair-PD^2 are that it calculates schedules for multiple time based task sets with arbitrary deadlines [59] like they occur

in automotive powertrain systems. In those systems, sources of task activation are either a (time based) periodic trigger, a crankshaft dependent trigger (based on the rotation speed of the engine), or a sporadic event trigger. The different sources of activation are described in Section 2.6.2. Further, Partly Pfair-PD^2 is a cooperative scheduler with non-preemptive task sections in order to minimize the amount of context switches. Finally, tasks have variable execution times during runtime which is the case in the most practical embedded real-time systems. The second algorithm is called Partly-Early Release Fair *P-ERFair* [62]. The algorithm is similar to Partly Pfair-PD^2 and applies the concept of partly Pfair onto the ERfair algorithm. The difference to Partly Pfair-PD^2 lies in the fact that pseudo activation is removed and thus P-ERFair is a work-conserving algorithm.

b) LLREF Scheduling Least Local Remaining Execution Time First scheduling (LLREF) which is optimal for periodic task sets with implicit deadlines was invented by [47]. It is based on an abstraction of time and local execution time domain plane (T-L plane) which was inspired by [57] and is non-work-conserving. Tokens represent the move of task over time. Figure 3.6 shows an example of such a T-L plane. In such a plane, the x-axis represents the time and the y-axis represents the remaining execution time of the task. The dotted line depicts the fluid schedule with a slope of -U(T_i). A task execution with a slope of -1 is drawn as line. If the line has a slope of zero, a non-execution task is represented. The figure shows the fluid schedules of N tasks. In this example, a right isosceles triangle for all tasks can be constructed between every two consecutive scheduling events.

The LLREF scheduling algorithm divides time-lines into sections separated by scheduling events. At the beginning of a section, LLREF selects m tasks according to their *least local remaining execution time*. While a task executes, its local remaining execution time decreases, whereas it remains constant for waiting tasks. LLREF inserts additional schedule events, whenever a task completes its local remaining execution time or a non-running tasks reaches the point where it has no laxity left. A work conserving extension of LLREF was provided by [70]. [71] presented an extension of LLREF, called LRE-TL. They made the observation that it is not necessary within each section to select tasks for execution based on their largest local remaining execution time. This

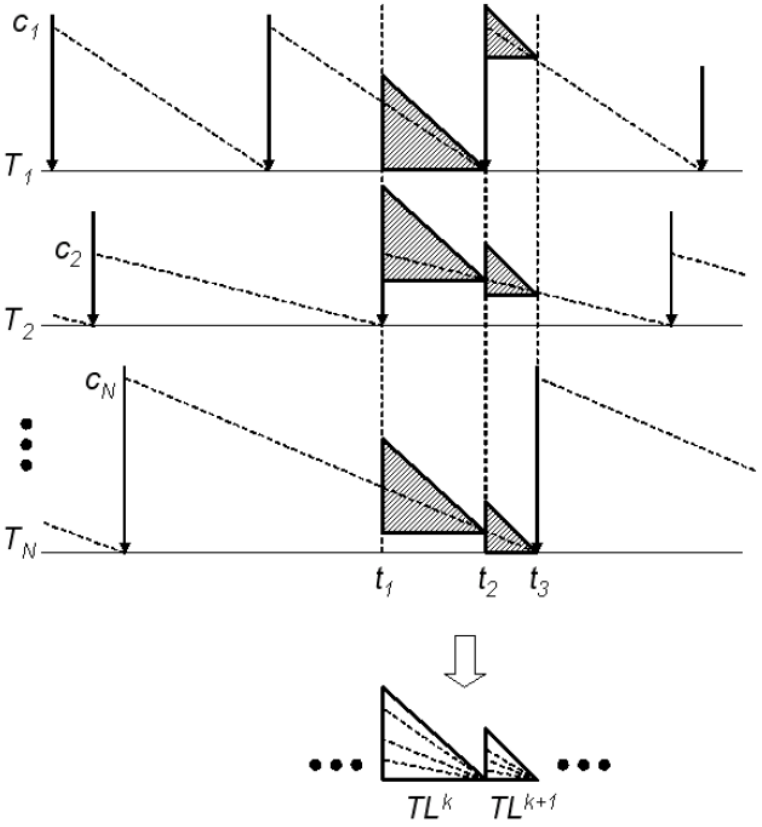

FIGURE 3.6: Example of a T-L plane for LLREF [47]. It shows a isosceles triangle between two consecutive scheduling events of the tasks and their fluid schedule.

extension reduces migrations by factor m and is shown to be optimal for sporadic task sets with implicit deadlines.

3.2.6 Clustered Multiprocessor Scheduling

Clustered Multiprocessor Scheduling is a combination of partitioned and global allocation of tasks to cores. m processors are divided into $\lceil \frac{m}{c} \rceil$ disjoint clusters [31].

Like under partitioned allocation, tasks are assigned statically to these clusters (jobs do not migrate across clusters). Within every cluster, a global scheduling algorithm is used to schedule jobs to the different cores belonging to the cluster. In contrast to partitioned scheduling, the bin-packing problem is simpler because there are fewer and larger bins of size c available while the utilization of tasks stays constant. Further, there is less implementation overhead compared to global scheduling because shared common cache can be used. Nevertheless, this approach assumes that the hardware architecture supports this approach (for instance the design of caches).

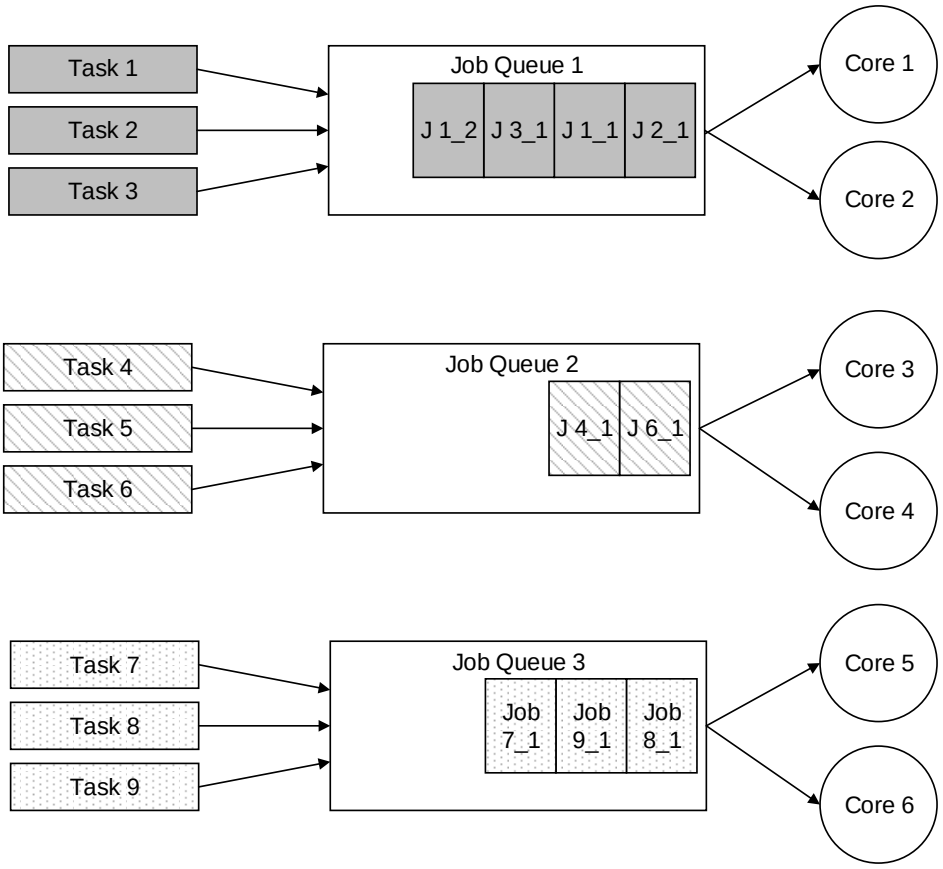

FIGURE 3.7: Example of clustered allocation of 9 tasks to 6 cores. A job $J_{i,j}$ of a task T_i is added on a job queue. Every job queue is shared by a cluster of cores and the scheduler decides which job is selected for which core of the corresponding cluster.

Generally, well-known algorithms can be used for clustered scheduling like it is presented in [39] where tasks are assigned statically to clusters and then scheduled by a global EDF scheduler.

A scheduling and model analysis was given by [127], where clusters are represented by multiprocessor periodic resource abstraction. Moreover, [127] presented an scheduling algorithm where clusters may be either physical with a static mapping or virtual with a dynamic mapping to processors. They have shown that this algorithm is optimal for task sets with implicit deadlines. However, in practice the high number of context switches that occur can be prohibitive [55].

An approach for scheduling mixed critical task sets (with hard and soft real-time requirements) was provided by [98]. It is a container based hierarchical attempt in which containers are allocated to a specific bandwidth provided by a periodic server.

3.3 Real-Time Synchronization

Schedulability analyses of scheduling algorithms that are described in the sections before are based on the assumption that tasks are independent. However, in practical systems this is not the case. That means that task have to share resources and many of them do not allow simultaneous access by different tasks. Access to shared resources within embedded real-time systems therefore has to be managed by a synchronization mechanism. Two different synchronization approaches exist for dependent tasks, lock-based and non-blocking synchronization mechanisms.

Non-blocking synchronization can further be divided into either *lock-free*, *wait-free* or *obstruction-free*. Lock-free synchronization means that system wide progress is guaranteed (at least one task is making progress), whereas wait-free ensures per-task progress (every operation has a maximum bound on the number of steps the algorithm takes before it is complete). Obstruction-free provides only weak progress guarantees (at any point, a single task executed in isolation for a bounded number of steps will be completed) and thus are not an option in systems with hard real-time requirements. However, non-blocking synchronization mechanisms usually require hardware support that often is not available in the observed embedded systems, additional memory or incur significant data copying or overhead by retry loops [31].

Therefore, we focus on the second approach, lock-based synchronization, in this dissertation.

In the following, a description about the fundamentals of lock-based synchronization is given. Further, problems that are accompanied with lock-based synchronization mechanisms are pointed out, as well as the necessary terminology and related work regarding uniprocessor and multiprocessor locking protocols is presented.

3.3.1 Fundamentals

At first we assume a system containing a set of shared resources R $\{R_1,...,R_k\}$ and a set of task τ $\{T_1,...,T_n\}$. When a task T_i wants to access a shared resource R_l it has to issue a request $Q_{i,l}$. If the resource is available, $Q_{i,l}$ is *satisfied* and thus the requesting task T_i enters a critical section. Otherwise, if R_l is not available, the request is *denied* and T_i has to wait until R_l becomes available. If a resource request $Q_{i,l}$ is denied, a task is

said to be *blocked*. Later, when task T_i has finished execution on the shared resource R_l, it gets released and thus the critical section is terminated. The released resource R_l is now available for other tasks again. Figure 3.8 shows an uniprocessor example schedule

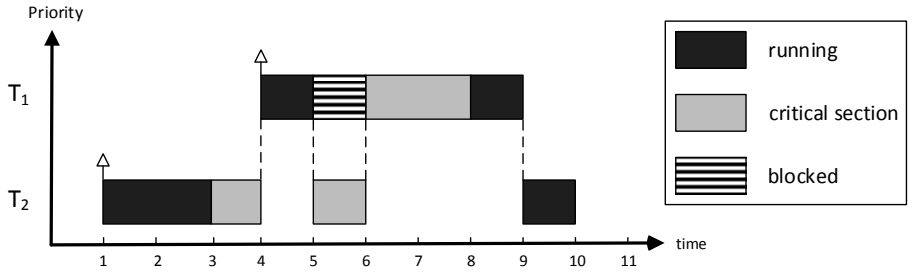

FIGURE 3.8: An example for blocking on a shared resource with mutual exclusion requirements (critical section). Task T_1 gets blocked at time 5 because task T_2 has acquired the shared resource already at time 3.

where a critical section occurs and a task gets blocked because its request is denied.

Task T_1 starts executing at time 1 and issues a resource request to a shared resource at time 3, which is satisfied (T_1 enters the critical section because the resource was acquired successfully). Task T_2 gets activated and preempts task T_2 because of higher priority. At time 5, task T_2 requires the same shared resource that is owned by T_1 at this point and issues a resource request. As this request is denied, task T_2 gets blocked and task T_2 is allowed to continue its critical section. After releasing the shared resource, task T_1 resumes (task T_2 gets preempted again).

3.3.1.1 Resource Models

This section can be seen as continuation of the definition of resources in Section 2.4. Different resource models (or sharing constraints) can be applied for shared resources. The most common one is mutual exclusion *mutex*. Resources with mutual exclusion can only be used by at most one task at any time. For resources with weaker constraints, *reader-writer exclusion* [52] may be a sufficient application. For this, it is required to be able to distinguish between read- and write accesses. Only reading a resource without affecting its value allows multiple read accesses, whereas write accesses have to be protected by mutex. In [33], different models of reader-writer exclusion are presented, for instance task-fair reader-writer locks or phase-fair reader-writer locks.

A third sharing constraint is the occurrence of x identical replicas, where each replica is only serially reusable and requires mutual exclusion, but up to x requests can be satisfied by accessing a own replica each. The approach of replica-exclusion is described in [27, 31] for example. If not described explicit differently, mutex resources are assumed within this dissertation.

3.3.1.2 Notation

For the calculation of the maximum blocking time of a task T_i, $B_{max}(i)$, some additional notations have to be defined.

Lock-based synchronization mechanisms modify priorities of tasks, following the goal of avoiding the later described synchronization problems. The nominal or either original priority Y_i of a task T_i denotes the priority of the task that is given according the applied scheduling policy. Further, the active priority y_i is dynamic and initially set to Y_i.

Next, $cs_{i,k}$ is a critical section of task T_i while executing on resource R_k, and $CS_{i,k}$ is the longest critical section of task T_i while executing on resource R_k. The duration of $CS_{i,k}$ is denoted by $\delta_{i,k}$. Finally, $\gamma_{i,a}$ is the set of longest critical sections that can block task T_i, while it is accessed by a lower priority task T_a, and $\sigma_{i,a}$ is the set of locks (e. g. semaphores) that can block T_i, while accessed by T_a.

3.3.2 Synchronization Problems

When tasks are synchronized with lock-based approaches, different problems might occur, as there were priority inversions, deadlocks, or starvation. These problems are discussed in the following.

3.3.2.1 Priority Inversions

Priority inversions take place if jobs are not running although they should be running in this morning. Consequently, the response time of those jobs gets exceeded. This could lead to unexpected deadline violations which is not acceptable in systems with hard real-time requirements. Figure 3.9 depicts an example schedule on a uniprocessor,

where a task suffers priority inversion.

Low priority task T_3 enters the critical section at time 2 but gets preempted by the high priority task T_1 at time 3. At time, task T_1 as well needs to access the critical section, but gets blocked as task T_3 owns the lock. Followed by this, task T_3 continues processing until time 5, where medium priority task T_2 gets activated and immediately preempts T_3 because of higher priority. This is the moment where priority inversion occurs because medium priority task T_2 is executing although high priority task T_1 is activated as well. After task T_2 terminates at time 7, task T_3 is resumed and leaves the critical section at time 8. Only then, task T_1 is allowed to enter the critical section and thus gets resumed. A famous practical example for priority inversion was yield by

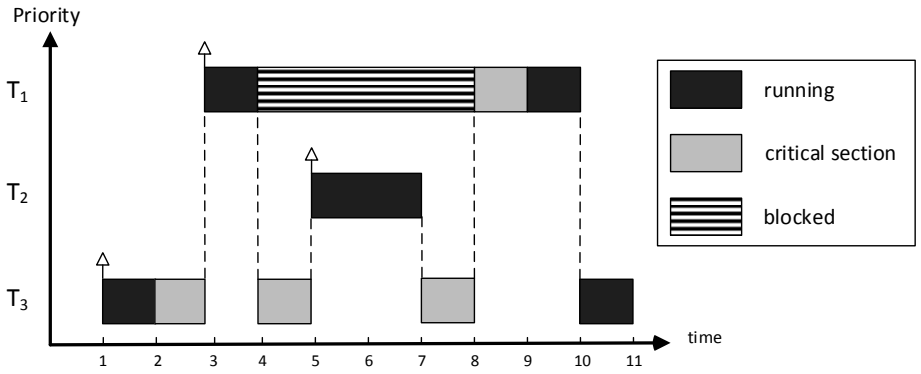

FIGURE 3.9: An example for priority inversion caused by resource sharing. Task T_2 starts executing although task T_1 has a higher priority at time 4.

the Mars Pathfinder mission that happened in july 1997. The Pathfinder mars robot experienced repeated resets by its watchdog process. The reason for that were timing overruns of some tasks caused by priority inversion. A software update that enabled priority inheritance (explained in Section 3.3.3.2) finally solved the problem.

3.3.2.2 Deadlocks

A deadlock occurs if two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. One can differentiate between the terms *deadlock* and *livelock*. A deadlock means that two or more processes remain in a fixed state while waiting for a reaction of the other process. In contrast, at a livelock, the involved processes frequently change their state, but they are not able to make any

progress. As deadlock and livelock lead to the same result regarding the system (no progress can be reached), we only refer to the term deadlock in the following. Such a behavior might only occur if lock-based synchronization mechanisms are used. Then a deadlock is possible, if and only if each of the following four conditions is fulfilled:

- *Mutual exclusion:* At least one resource can be used by only one single process at a time.
- *Hold and wait:* One process is holding a resource and waiting for another resource simultaneously.
- *No preemptions:* Critical sections cannot be preempted, resources can only be released by the process that is holding it.
- *Circular wait:* There exist a cycle of process Requests, for instance a set of processes τ T_1, T_2, T_3,...,T_n such that T_1 is waiting for a resource which is held by T_2 and T_2 is waiting for $T_3, ...$ and T_n waits for T_1.

The following example in Figure 3.10 describes a deadlock scenario for two tasks on a uniprocessor which is explained in [38].

At time 2, the low priority task T_2 acquires resource *b*. Task T_2 gets preempted at

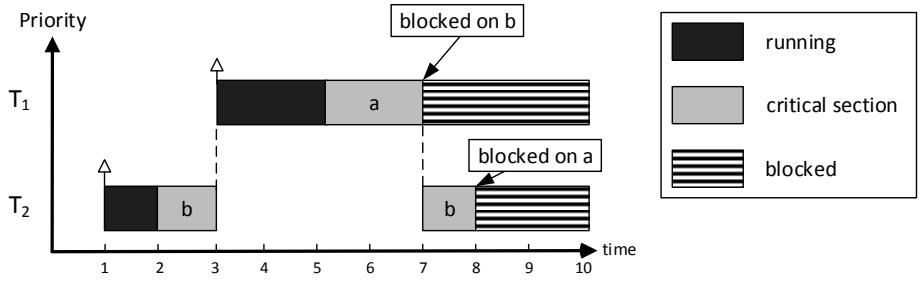

FIGURE 3.10: An example for a deadlock. Task T_1 waits for task T_2 to release resource b, while task T_2 waits for task T_1 to release resource b.

time 3 by the high priority task T_1. While task T_1 runs, it acquires resource *a* at time 5 and further tries to acquire resouce *b* but gets blocked, because resource *b* is already kept by task T_2. Followed by this, task T_2 gets resumed and executes in critical section with resource *b* until time 8 where it tries to lock resource *a*. After time 8, both tasks are waiting for each other to release the acquired resources and no task is able to make

progress in the sense of executing further instructions. A deadlock occurred. Note, that a livelock is similar to a deadlock, except that tasks are doing useless work instead of waiting, for instance if tasks permanently issue retry-loops that never are terminated successfully.

3.3.3 Uniprocessor Real-Time Locking Protocols

Locking protocols are required to overcome the synchronization problems of deadlocks (described in Section 3.3.2.2) and unbounded priority inversions (discussed in Section 3.3.2.1). Note that deadlock avoidance requires some kind of priority inversions, but locking protocols should bound and minimize them. For uniprocessor systems, well studied and optimal locking protocols exist. Some of these protocols are discussed within the following sections.

3.3.3.1 Non-Preemptive Critical Sections

The simplest approach to avoid deadlocks and unbounded priority inversions is to make critical sections non-preemptive. That means, as soon as a job has acquired a resource successfully, it becomes non-preemptive immediately for all the other jobs, no matter if those other jobs have a higher priority or not. In this dissertation we refer to this protocol as *non-preemptive protocol* (NPP). Figure 3.11 shows an example schedule of the NPP which is explained in the following. The low priority task T_3 enters a critical

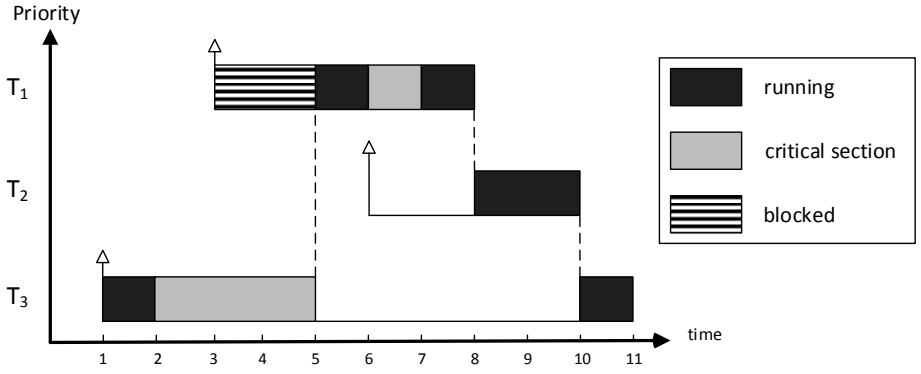

FIGURE 3.11: Example schedule with non-preemptive protocol. Task T_3 becomes non-preemptive as it enters its critical section. Followed by this, task T_1 gets blocked although it has a higher priority.

section at time 2 and thus gets non-preemptive. Further, at time 3, the high priority task T_1 gets activated, but is not allowed to execute because task T_3 is non-preemptive at this moment. After task T_3 has left its critical section at time 5, task T_1 gets resumed and preempts task T_3. Task T_1 enters a critical section at time 6 and gets non-preemptive as well, but this does not influence the schedule in this example because task T_1 is the task with the highest priority within this task set anyhow. The set of longest critical sections that can block task T_i is given by:

$$\gamma_i = max\{CS_{a,k} \mid Y_a < Y_i,\ k = 1, ..., m\} \tag{3.13}$$

Because of the fact, that task T_i can not be preempt a lower priority task T_y that is inside a critical section, it follows that task T_i can be blocked by any critical section inside a lower priority task. The maximum blocking duration of the NPP can then be calculated as:

$$B_{max}(i) = max\{\delta_{a,k} - 1 \mid CS_{a,k} \in \gamma_i\} \tag{3.14}$$

One unit of time has to be subtracted because a task T_i can only be blocked by a lower priority task T_y if it has already started its critical section cs_y. The NPP is widely used in many practical systems because it is easy to implement (it is just necessary to disable interrupts during critical sections) [31]. Nevertheless, the NPP contains the drawback that jobs that do not access any shared resources suffer from blocking (priority inversion) and thus might unnecessarily violate their deadlines.

3.3.3.2 Priority Inheritance Protocol

The priority inheritance protocol (PIP) was presented by [125]. It is designed for task sets with task-fix priorities. PIP avoids the described disadvantage of the NPP that jobs which do not access any critical sections are blocked unnecessarily. At the PIP, blocking only occurs if the preemption of a resource holding, lower priority job would cause a delay for a higher priority job. Lower priority jobs that are inside a critical section inherit priority of the highest priority waiting job. Followed by this, only jobs with a higher priority than the highest priority of waiting jobs are able to preempt the job within a critical section. Figure 3.12 shows the following example schedule with PIP:

The low priority task T_3 enters a critical section at time 2. At time 3, high priority task

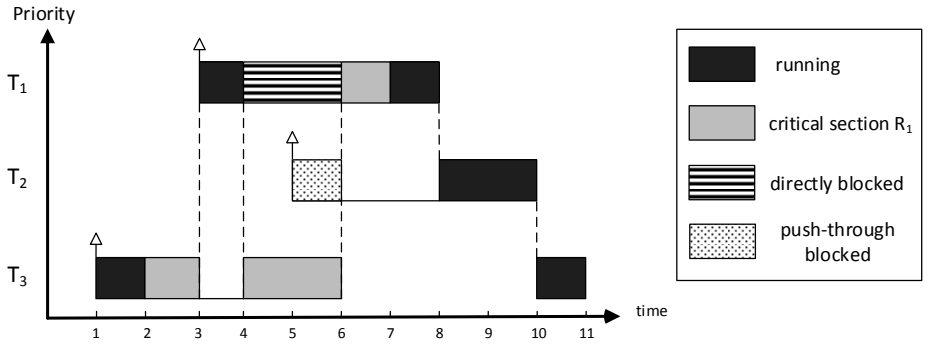

FIGURE 3.12: Example schedule with priority inheritance protocol. Task T_3 inherits the priority of task T_1 at time 4 until it has finished its critical section at time 6. Different types of blocking occur in this example.

T_1 gets activated and preempts task T_3 because of higher priority. Later, at time 4, task T_1 tries to access the shared resource that has already been acquired by task T_3. According to the rules of PIP, task T_3 inherits the priority of task T_1 at this point and gets resumed for the residual duration of the critical section. At time 6, task T_3 leaves the critical section and gets back its original priority, which leads to a preemption by the high priority task T_1 that is now able to enter the critical section. Unbounded priority inversion is avoided by PIP which can be observed at time 5 when medium priority task T_2 gets activated, but low priority task T_1 continues executing because it has inherited task T_1's priority. After that, when task T_1 terminates at time 8, task T_2 is allowed to execute because of higher priority than task T_3 which has its original priority at this point in time.

Figure 3.10 shows that tasks can experience two different kinds of blocking. *Direct Blocking* occurs if a higher priority task gets blocked by a low priority task because it wants to enter a critical section that has already been entered by the low priority task. This blocking is required to keep data consistency [38]. In contrast, *Push-Through Blocking* means that a medium priority task is blocked because a low priority task has temporarily inherited the priority of a high priority task. Push-though blocking is required to avoid unbounded priority inversions.

The evaluation of blocking times is quite difficult for PIP because of the different types of blocking that may occur. A precise deviation of this analysis can be found in [38], it

results in two terms of blocking:

$$B^{l}_{max}(i) = \sum_{j:Y_a<Y_i} \max_{k}\{\delta_{a,k} \; - \; 1 \mid CS_{a,k} \; \in \; \gamma_i\} \tag{3.15}$$

$$B^{s}_{max}(i) = \sum_{k=1}^{m} \max_{a}\{\delta_{a,k} \; - \; 1 \mid CS_{a,k} \; \in \; \gamma_i\} \tag{3.16}$$

The maximum overall blocking time $B_{max(i)}$ is:

$$B_{max}(i) = min(B^{l}_{max}(i), B^{s}_{max}(i)) \tag{3.17}$$

Where the set of longest critical sections is:

$$\gamma_{i,a} = \{CS_{a,k} \mid (Y_a < Y_i) \; and \; (R_k \in \sigma_{i,a})\} \tag{3.18}$$

Followed by this, the set of all critical sections that may block T_i directly or by push-through is:

$$\gamma_i = \bigcup_{j:P_a<P_i} \gamma_{i,a} \tag{3.19}$$

Unfortunately, PIP does not guarantee deadlock freedom. The deadlock example (Figure 3.10) in section 3.3.2.2 would also occur in the same way if PIP is applied.[3] A further disadvantage of PIP is that there occurs a high number of context switches which causes additional overhead.

3.3.3.3 Priority Ceiling Protocol

The priority ceiling protocol was presented (PCP) by Sha et al and Rajkumar respectively [125, 119]. Like PIP, the PCP is designed for task sets with task-fix priorities. In contrast to PIP, PCP is deadlock free because a task is only allowed to enter a critical section if it is free and there exists no risk of chained blocking.

Each resource R_k is assigned a ceiling $\Pi(R_k)$ which denotes the maximum priority Y_max of all tasks $T \in \tau$ that can lock R_k (can be calculated off-line). A task T_i is only allowed to enter a critical section if its priority is greater than the ceilings of all resources that are currently locked by other tasks. When a task T_i is blocked, it transmits its priority to the resource holding task. Followed by this, the task that has acquired the lock resumes

[3]Note that deadlocks are only possible if nested critical sections occur.

and executes the remaining part of its critical section. After the release, the releasing job regains its original priority. An example schedule is shown in Figure 3.13. The low

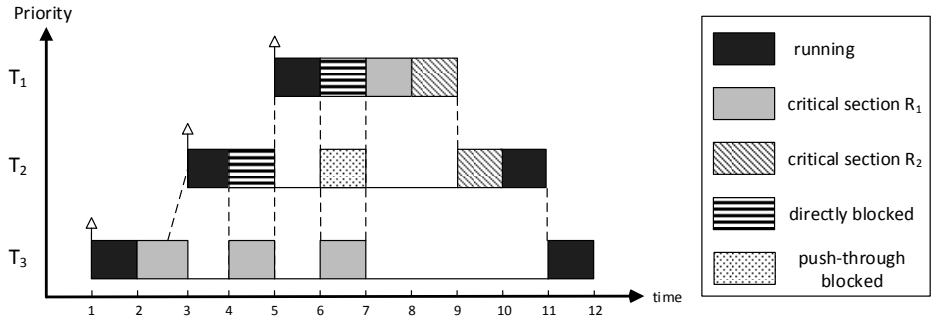

FIGURE 3.13: Example schedule with priority ceiling protocol. Task T_1 and task T_2 getting blocked at time 4 and time 6 respectively because their priority does not lie above the ceiling of resource R_1 that is acquired by task T_3.

priority task T_3 enters a critical section (resource R_1) at time 2. At time 3, medium priority task T_2 preempts task T_3 because of higher priority. Further, at time 4, task T_2 tries to enter another critical section (resource R_2). However, as high priority task T_1 accesses both resources as well, task T_3 gets the ceiling priority and continues execution within its critical sections. The same proceeding happens with task T_1 at time 6, where PCP avoids unbounded priority inversion (task T_2 gets push-through blocked). After task T_3 has released resource R_1, it returns to its original priority and gets preempted by the task which has the highest priority of all active tasks (task T_1).

Contrary to the NPP, a task T_i can only be blocked by critical sections that belong to lower priority tasks with a resource ceiling that is greater or equal than Y_i:

$$\gamma_i = max\{CS_a \mid (Y_a < Y_i)\ and\ \Pi(R_k) \geq Y_i\} \tag{3.20}$$

The blocking time is calculated as follows:

$$B_{max}(i) = max\{\delta_{a,k}\ -\ 1 \mid CS_{a,k}\ \in\ \gamma_i\} \tag{3.21}$$

The PCP avoids deadlocks, chained blocking and unbounded priority inversions. However it has the disadvantages that PCP leads to many context switches, and further it is difficult to implement. Each semaphore has to store its ceiling and further the active priorities of tasks have to be stored in the task control block [38].

3.3.3.4 Highest Locker Protocol

The highest locker protocol (HLP), which is also known as immediate priority ceiling protocol (IPCP), was proposed by [90]. The concept of HLP is similar to the original PCP, the main difference lies in the point of time where the blocking takes place. A job can only be blocked at release time, otherwise it does not get blocked at all. When a blocked job later requests a resource, it is satisfied immediately (per definition). All resources that j might request are available after the blocking at release time. As a consequence, deadlocks are not possible.

The following example schedule of HLP is drawn in Figure 3.14. The low priority task T_3 enters a critical section (resource R_1) at time 2. At time 3, medium priority task T_2 gets activated with a higher priority than the running task T_3, but task T_3 continues executing because it is within a critical section and the HLP rule is that a task can only be blocked at release time. Further, at time 5, task T_3 leaves the critical section and thus gets preempted by task T_1 (T_1 is activated at time 5 and has higher priority than task T_2). The set of longest critical sections that can block a task T_i is equal to that

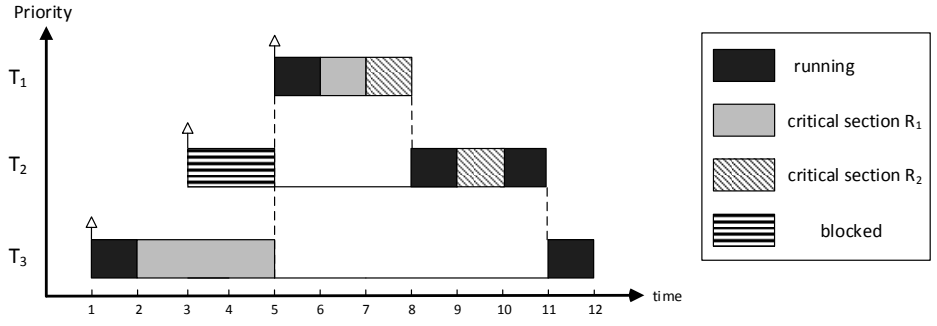

FIGURE 3.14: Example schedule with stack resource policy. Task T_2 gets blocked immediately at its activation (time 3) by lower priority task T_3, although it requests resource R_2 at a later point in time.

set in PCP:

$$\gamma_i = max\{CS_a \mid (Y_a < Y_i)\ and\ \Pi(R_k) \geq Y_i\} \tag{3.22}$$

Moreover, it is not very surprising that the maximum blocking time is the same as in PCP as well, since the only difference between these two protocols is the point in time

where blocking occurs:

$$B_{max}(i) = max\{\delta_{a,k} \; - \; 1 \mid CS_{a,k} \; \in \; \gamma_i\} \tag{3.23}$$

The HLP has several advantages. Like the PCP it avoids deadlocks, chained blocking and unbounded priority inversions. Further, context switches are reduced and it is easier to implement in comparison to PCP. Note that up to this point all presented locking protocols were built on the assumption that task-fix priority scheduling is applied. However, the NPP, PIP and PCP can be extended in a way that they can be applied under job-fix priority scheduling as well [31]. The HLP can be seen as a specific case (for task-fix priorities) of the stack resource policy which is presented in the following.

3.3.3.5 Stack Resource Policy

Stack resource policy (SRP) was presented by Baker [16] and supports task-fix as well as dynamic priority scheduling. It is an extension of PCP that allows multi-unit resources. Besides the resource ceiling Π (of every single resource), it further requires a system ceiling $\hat{\Pi}$. The SRP rule is that a job can not preempt another job until it has the highest priority of all activated jobs and further its resource ceiling is greater than the system ceiling.

SRP uses the same blocking scheme as the HLP in section 3.3.3.4, where jobs can only be blocked at release time. Followed by this, the example schedule of the HLP in Figure 3.14 is valid for SRP as well.

The set of lower priority tasks that can block higher priority tasks while being in a critical section can be extended from the calculation of γ at the PCP. In contrast to PCP, the SRP does not work with static (task-fix) priorities, but rather with so called preemption levels π. A task T_i is only allowed to preempt another task T_a, if $\pi_i > \pi_a$, and further if T_i arrives after T_a and T_i has a higher priority than T_a. Then follows $\pi_i > \pi_a$. γ_i within SRP can be calculated as:

$$\gamma_i = max\{CS_a \mid (\pi_a < \pi_i) \; and \; \Pi(R_k) \geq \pi_i\} \tag{3.24}$$

The calculation of the maximum blocking time bound is the same as for HLP and PCP:

$$B_{max}(i) = max\{\delta_{a,k} \ - \ 1 \mid CS_{a,k} \ \in \ \gamma_i\} \tag{3.25}$$

SRP as well avoids deadlocks, chained blocking and unbounded priority inversions. Further advantages are that it is easy to implement and allows sharing of stack-based resources.

3.3.3.6 Bandwidth Inheritance Protocol

The bandwidth inheritance protocol (BIP) was published by [96]. Its blocking behavior is similar to that from the PIP, but it uses a server-based approach to realize the blocking mechanism. Followed by this, it can be considered as an extension of PIP to resource reservation. Instead of inheriting just the priority, the lock holding task inherits the entire reservation of each task that attempts to acquire the resource (a deviation of the constant bandwidth server principle). A further difference to all the other presented protocols is that the BIP is built for open systems, where tasks can be added or taken away from the task set at runtime. However, the BIP is difficult to implement, and as we do not consider such open systems, we do not analyze this protocol more in detail.

3.3.4 Multiprocessor Real-Time Locking Protocols

Uniprocessor protocols are not longer sufficient when multiprocessors are applied, since resources may be requested by jobs on different cores simultaneously. Thus, special multiprocessor locking protocols are required to avoid deadlocks and unbounded priority inversions. We continue with a description of the different categories, followed by a detailed discussion of existing real-time locking protocols for multiprocessors.

3.3.4.1 Categories

Multiprocessor locking protocols can be classified into different categories. Table 3.2 gives an overview of these categories, which are explained in detail afterwards.

TABLE 3.2: Classification of multiprocessor real-time locking protocols.

	I	II	III	IV
Blocking	Busy-Wait	Suspend		
Nesting	Yes	No		
Priorities	Task-fix	Job-fix	Section-fix	Dynamic
Allocation	Partitioned	Global	Clustered	

a) Blocking The first category - blocking - can be performed either by busy-wait or by suspending. Busy-wait means that tasks whose resource requests are denied and thus get blocked, stay active on their assigned processor while waiting. If this scenario occurred in uniprocessors, the system would starve.

However, there exist different types of busy-wait blocking. Blocked tasks can wait either preemptively or non-preemptively, and further the critical section as itself can be performed preemptive or non-preemptive as well. Besides, wait queues (where blocked tasks are listed) may be sorted in FIFO manner, by priority, in round-robin manner or just be unordered [4]. An exact description of the different types of busy-wait blocking can be found in [138]. The drawback of busy-wait blocking is that processor time is wasted as blocked tasks stay active but do not make any progress.

The example non-preemptive busy-wait scenario depicted in Figure 3.15 shows a schedule of 3 tasks on 2 processors. Task T_3 which runs on processor P_2 enters its critical section (Resource R_1) at time 2. At time 3, task T_2 which runs simultaneously on processor P_1 issues a request Q_1 for R_1 as well and gets blocked because task T_3 already owns resource R_1. Followed by this, T_2 busy-waits and stays active on P_1, even if a higher priority task T_1 gets activated. After T_3 has released R_1 at time 5, T_2 enters its critical section. Finally, at time 6, T_2 releases R_1 as well and T_1 gets scheduled. One advantage of the busy-wait approach is that the number of context switches and preemptions are minimized.

Secondly, a suspension based protocol preempts tasks whose resource requests are denied in order to let execute other tasks without resource conflicts.

Figure 3.16 shows the same scenario from Figure 3.15, but now suspension is applied. As T_2 gets blocked because of the denied resource request at time 3, it suspends. Now,

[4] Usually, each shared resource has its own wait queue. The resource is assigned to the task that is the head of this queue.

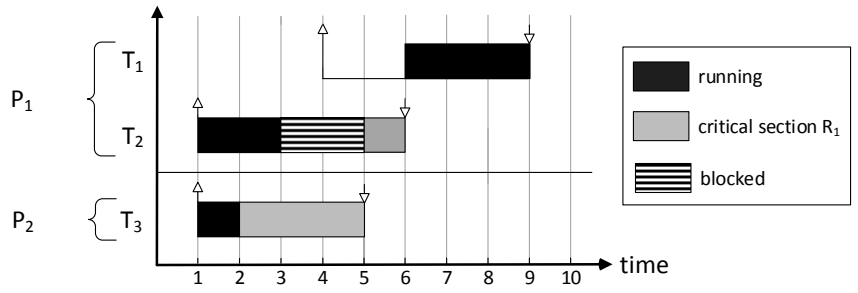

FIGURE 3.15: Example schedule of 3 tasks on 2 processors. Task T_2 gets busy-wait blocked by task T_3 at time 3 until time 5. It stays active on processor P_1 and does not let execute the higher priority task T_1 at time 4.

T_1 gets scheduled and runs until it gets terminated at time 7. After that, T_2 continues execution and successfully acquires resource R_1. Suspension based protocols often

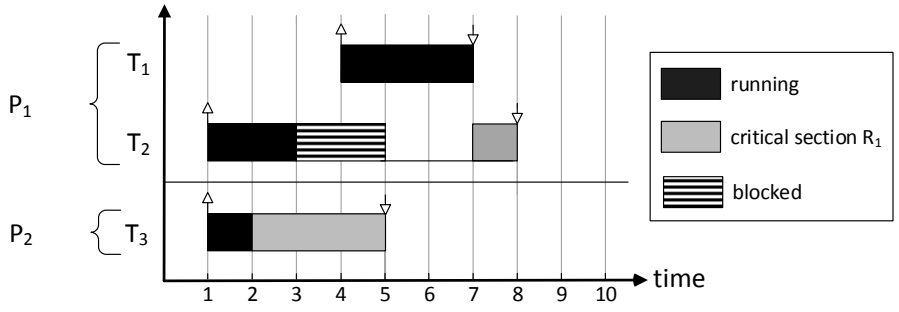

FIGURE 3.16: Example schedule 3 tasks on 2 processors. Task T_2 gets blocked by task T_3 and suspended. When higher priority task T_1 gets activated at time 4 it starts to execute.

bound the number of suspensions by either priority boosting, priority donation [28] or priority inheritance [137]. Priority boosting means that a task gets it priority raised unconditionally in order to ensure that it gets scheduled. Priority donation occurs if a higher priority task T_i, that should preempt a lower priority task T_a with a denied resource request $Q_{a,l}$, suspends and instead donates its priority to T_a and thus ensures that task T_a makes progress [28]. Finally, priority inheritance means that a task T_i executes with the priority of a suspended task T_a.

An example for priority boosting is given in Figure 3.17, where task T_2 gets priority boosted after getting suspended so that it gets scheduled as soon as T_3 has released R_1. The advantage of suspension-based protocols is that other tasks which are not blocked

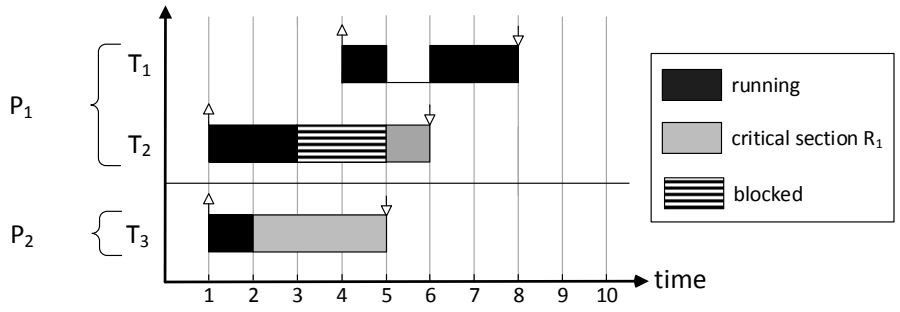

FIGURE 3.17: Example schedule 3 tasks on 2 processors. Task T_2 gets blocked by task T_3 and suspended with priority boosting. When task T_3 leaves its critical section at time 5, task T_2 gets priority boosted until it releases the critical section at time 6.

are able to execute while blocked tasks are suspended. The drawback of suspension is that many preemptions and context switches occur. The most important design decision of a multiprocessor locking protocol is, whether it should be either busy-wait or suspension based. As a consequence, we distinguish the later described protocols (in section 3.3.4.2 and following) in terms of blocking.

b) Nesting Nesting means that a task can issue a resource request $Q_{i,k}$ for a resource R_k, even if it has already acquired a resource R_l via a former resource request $Q_{i,l}$. That means that tasks are allowed to own more than one resource simultaneously. If a multiprocessor locking protocol allows nesting, the deadlock avoidance is more difficult than without nesting.

The example in Figure 3.18 shows a task that issues nested resource requests. At time 2, it requests resource R_a via Q_a, and enters the corresponding critical section as no other task has acquired resource R_a. Later, at time 3 the task further issues a request Q_b for resource R_b, which is a nested request. The task is allowed to access the critical section of R_b as well, so it owns both, resource R_a and resource R_b simultaneously. In this scenario, the resource request Q_b is said to be nested within resource request Q_a.

c) Priorities The different types of priorities are equal to the ones in table 3.1 at point Prioritization. Generally, not all multiprocessor locking protocols are able to be applied under any scheduling policy. For instance, protocols based on priority ceiling need to

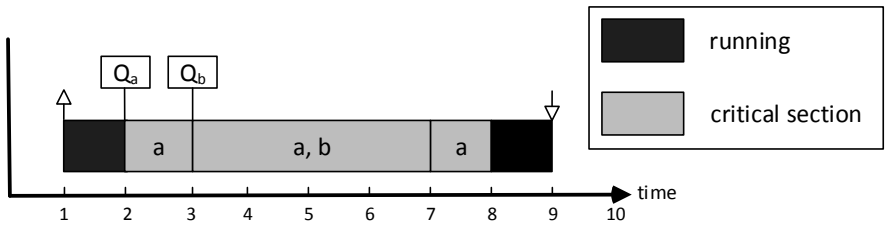

FIGURE 3.18: Example schedule of a task with nested resource requests Q_a and Q_b. CS_b is nested within CS_a.

work with task-fix priorities. Nevertheless, there exist protocols that can be combined with any kind of priorities, task-fix, job-fix, section-fix or dynamic tasks priorities.

d) Allocation Some of the protocols distinguish between local and global resources, where local resources are only accessed by tasks running on the same processor, whereas global resources can be accessed by tasks on any processor. For local resources, also the presented uniprocessor locking protocols can be used for synchronization. Protocols that are applied under a global scheduling algorithm, all resources have to be global obligatory because tasks can be executed on any core and thus resource requests can be made from any core as well.

3.3.4.2 Suspension Based Synchronization Protocols

The first multiprocessor real-time locking protocols, the distributed priority ceiling protocol D-PCP [120, 119] and the multiprocessor priority ceiling protocol M-PCP [118, 119] are both suspension-based. Both protocols are extensions of the PCP under partitioned task-fix priority scheduling. The following sections discuss these two protocols, as well as further suspension-based protocols that were invented later.

a) Distributed Priority Ceiling Protocol D-PCP The distributed priority ceiling protocol D-PCP was developed for application in distributed memory multiprocessors. Initially it was named multiprocessor priority ceiling protocol M-PCP in [120] and renamed to D-PCP in [119]. This might be confusing because the M-PCP is an own protocol for shared memory multiprocessors which is described in Section 3.3.4.2.

A result of the assumption that distributed memory multiprocessors are used, each resource is only accessible on a specific processor. However, D-PCP distinguishes between local and global resources. Local resources are only accessed by tasks running on the same processor and handled with the uniprocessor PCP policy. In contrast, global resources R_l are assigned to a *local agent* A_l each. If a global resource R_l has to be accessed by a task T_i which is not running on the same core as its local agent A_l, a so called *remote procedure call* RPC is issued by the requesting task T_i. The RPC calls the local agent A_l of the requested resource R_l. A_l then becomes active on its processor. Local agents carry out requests that are issued from another core after they receive a call via RPC. After T_i has issued a request $Q_{i,l}$ and thus called A_l, it suspends until it is resumed when csi, l is terminated within A_l. Executions of requests to global resources are priority boosted, while the priorities of requesting tasks remain unchanged. Agents access global resources on their core according PCP rules as well. Figure 3.19 presents

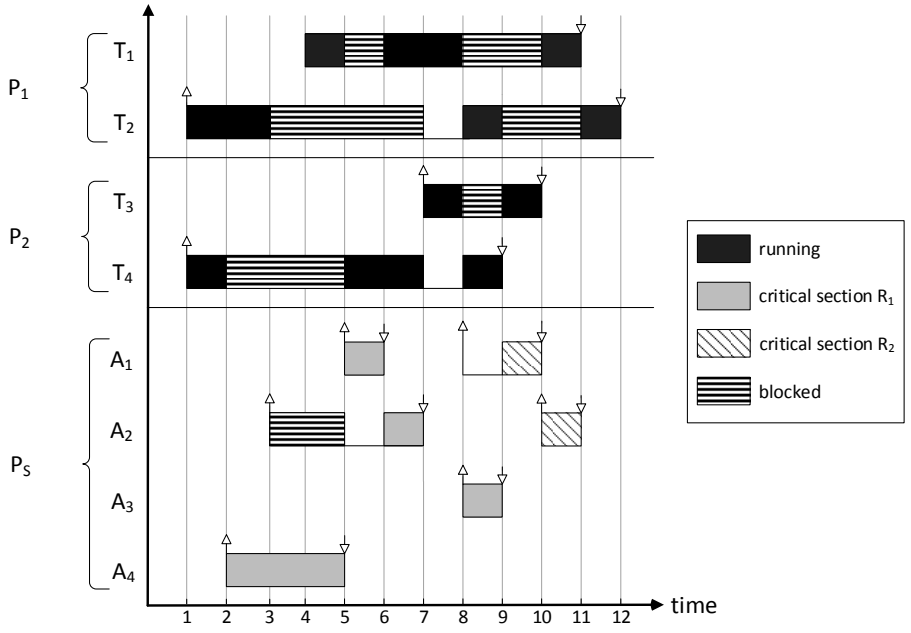

FIGURE 3.19: Example schedule with distributed priority ceiling protocol. Global resources are accessed via agents on a synchronization processor P_s. Agents are executed according to PCP rules. Tasks on processors P_1 and P_2 become suspended while waiting for agents to terminate their critical sections.

an example schedule of the D-PCP with four tasks (T_1 - T_4) allocated to two processors (P_1 and P_2), and two global resources (R_1 and R_2) allocated to a synchronization processor P_S. Global resources can be accessed by corresponding agents (A_1 - A_4) that

are called by the requesting tasks per RPC. Partitioned, task-fix priority scheduling is applied as it is necessary for the D-PCP.

At time 2, T_4 issues a resource request $Q_{4,1}$ to the global resource R_1 and thus calls its corresponding agent A_4 on the synchronization processor. Agent A_4 carries out the critical section for task T_4, while task T_4 gets suspended. Further, at time 3, task T_2 issues a resource request $Q_{2,1}$ to resource R_1 as well. Agent A_2 is called, but gets blocked because agent A_4 executes the critical section on resource R_1 at this point in time. Nevertheless, task T_2 gets suspended on processor P_1 and allows task T_1 which is activated at time 4) to start its execution meanwhile. When resource R_1 is released by agent A_4 at time 5, there are two agents activated (A_1 and A_2). A_1 starts the critical section at this point because agent A_1 has a higher priority than agent A_2 (analog to the priorities of their corresponding tasks T_1 and T_2). Further, as agent A_4 is terminated at time 5 as well, task T_4 gets resumed and continues execution on processor P_2. The remaining critical sections are performed according to the rules of partitioned, task-fix priority scheduling. For reasons of simplicity, accesses to local resources, which would be handled according to the uniprocessor PCP, are not considered within this example.

An advantage of D-PCP in terms of blocking analysis is that resource requests are executed on remote processors and do not influence the execution time and utilization of tasks. Blocking analyses for the D-PCP can be found in [120, 119, 102].

b) Multiprocessor Priority Ceiling Protocol M-PCP The multiprocessor priority ceiling protocol M-PCP [118, 119] is a later version of D-PCP for shared memory multiprocessors. In contrast to D-PCP where distributed memory multiprocessors are assumed, each resource can be accessed by any processor within the M-PCP. That means that no local agents and thus no RPCs are required anymore. A task T_i that issues a request $Q_{i,l}$ for a global resource R_l instead accesses R_l directly and is priority-boosted while the critical section $cs_{i,l}$ is in progress. The boosted priority is the highest priority of any remote task T_a that shares the requested resource R_l.

Since remote priorities are not necessarily unique, tie-breaking rules are required. The first rule is that earlier boosted jobs have precedence to avoid unnecessary preemptions. Further, if there exist tasks that have the same (boosted) priority and were boosted at the same point in time the basic priority of job is the second tie-breaking criterion. The example depicted in Figure 3.20 shows the same example of Figure 3.19, but with the

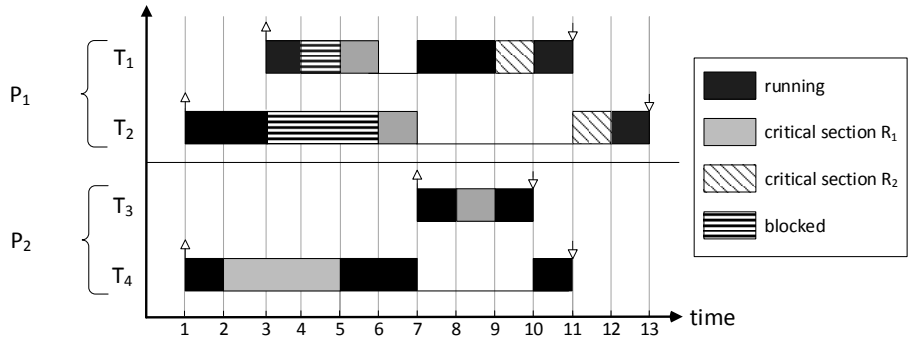

FIGURE 3.20: Example schedule with multiprocessor priority ceiling protocol. Global resources are accessed directly by the requesting tasks. Tasks on processors P_1 and P_2 are suspended while waiting for the release of requested resources.

M-PCP applied instead of D-PCP. The main difference is that M-PCP assumes that shared memory is used. Task T_4 requests resource R_1 and executes its critical section from time 2 to time 5. Meanwhile, at time 3, task T_2 issues a request to resource R_1 as well and gets blocked (suspended from processor P_1). Task T_2 is thus remote blocked by T_4. Later, as task T_1 is activated at time 3, it gets active and starts execution. Further, T_1 enters the critical section with resource R_1 at time 5 (the request was issued at time 4) because it has a higher (boosted) priority than task T_2. A detailed blocking aware response time analysis for the M-PCP can be found in [95].

c) Multiprocessor Dynamic Priority Ceiling Protocol M-DPCP Both protocols, D-PCP and M-PCP, were extended by [45] to M-DPCP I and M-DPCP II. They developed versions for these two protocols that are able to handle task sets with job-level priority partitioned scheduling. They use non-preemptive critical sections for synchronization and allow nesting of resource requests. Multiple requests to a resource are handled by a priority ordered wait-queue. Finally, they showed a blocking time analysis.

d) O (m) Locking Protocol OMLP A new family of locking protocols was presented by [31, 27], where protocols were built for either mutex, reader-writer exclusion, and k-exclusion. These protocols are all suspension-based, allow nesting and support partitioned, as well as global and clustered scheduling. However, it was mainly designed for application at clustered scheduling. Multiple requests to resources are handled by

a FIFO queue per resource and priority donation is used to bound suspensions. If a task T_i issues a request $Q_{i,l}$ to a resource R_l and R_ls FIFO queue is non-empty, the requesting task T_i suspends until $Q_{i,l}$ is granted ($Q_{i,l}$ is head of R_ls FIFO queue). Further, if R_ls FIFO queue is full, further requests are inserted into an additional priority sorted wait-queue. A detailed blocking analysis can be found in [31]. It shows that the maximum bound a task T_i can be (priority-inversion) blocked is calculated by:

$$B_{max}(i) = \sum_{k=1}^{n_r} N_{i,l} \cdot (2m-1) \cdot CS^{max} \tag{3.26}$$

Where m denotes the number of processors, n the number of running tasks, CS^{max} denotes the maximum critical section length and $N_{i,q}$ is the maximum number of times that any T_i requests R_l. One drawback of this family of protocols is the implementation difficulty, which requires token-based locks and further priority inheritance in case of global scheduling and a type of priority boosting for partitioned and clustered scheduling.

e) FIFO Multiprocessor Locking Protocol FMLP$^+$ The FIFO multiprocessor locking protocol FMLP$^+$ provided in [31] is developed for application of partitioned scheduling. Lock-holding tasks are priority boosted. That means that they get higher priority than tasks that do not hold resources. Priority boosted jobs are ordered by the point in time where the resource request was issued (FIFO). Later, in [32], the FMLP$^+$ was extended for global and clustered job-fix priority scheduling. Within the extension a restricted segment priority boosting is applied, where at most one lock-holding task gets priority boosted, with the intention to disturb the general schedule as little as possible. A blocking analysis of the FMLP$^+$ can also be found in [32]. For the case that critical sections are preemptive:

$$B_{max}(i) = (n_c - 1) \cdot CS^{max} + \sum_{l=1}^{n_r} N_{i,l} \cdot (n-1) \cdot CS^{max} \tag{3.27}$$

And further, if critical sections are performed non-preemptive:

$$B_{max}(i) = (n_c - 1) \cdot \left(1 + \sum_{l=1}^{n_r} N_{i,l}\right) \cdot CS^{max} + 2 \left(\sum_{l=1}^{n_r} N_{i,l} \cdot (n - n_c)\right) \cdot CS^{max} \tag{3.28}$$

f) Multiprocessor Synchronization Protocol For Real-Time Open Systems MSOS The Multiprocessor Synchronization Protocol For Real-Time Open Systems MSOS is a family of locking protocols. They are built for real-time open systems, where applications can be added and removed dynamically from the task set. Applications consist of a certain amount of tasks. Resources are divided into local and global resources. Local resources belong to one single application, whereas global resources are shared between different applications. The first MSOS protocol, MSOS-FIFO [113] is built for partitioned scheduling and assumes that an application is fix assigned to a certain processor without a possibility for migration. Thus, local resources can be synchronized by any uniprocessor locking protocol. Each global resource has its own FIFO queue, where requests from the different applications are enqueued. That application that is the head of the queue is allowed to access the resource. However, each application has a FIFO queue as well, where tasks waiting for resources are listed. If the resource becomes available for an application its task that is head of the local application FIFO queue is granted access to the resource. Tasks that are inside a critical section are priority boosted in order to ensure that resources are released as quickly as possible.

A second MSOS protocol, MSOS-Priority [111] is similar to MSOS-FIFO but wait queues are sorted by priority and not by time stamp of requests.

Finally, C-MSOS [112] is developed for clustered scheduling algorithms. Under C-MSOS, local resources are handled by PIP. The wait queues of global resources are sorted either in FIFO or in round robin manner. In contrast to MSOS-FIFO and MSOS-Priority, local queues exist not per application, but per cluster. Priority boosting is applied inside each cluster.

3.3.4.3 Busy-Wait Based Synchronization Protocols

The second type of synchronization protocols are busy-wait based. Busy-wait means that if a task T_i issues a request $Q_{i,l}$ to a resource R_l and it is denied, the blocked task becomes not suspended, but rather it actively waits until R_l becomes available again. Since blocked tasks are still in state running while waiting for resources, they are waisting processor time (these tasks make no progress during busy-wait). This behavior is a disadvantage compared to suspension based protocols, whereas an advantage is that

busy-wait based protocols minimize overhead, namely context-switches and migration. Different busy-wait based synchronization protocols are discussed in the sections below.

a) Multiprocessor Stack Resource Policy M-SRP Multiprocessor stack resource policy M-SRP was presented by [73]. M-SRP is based on uniprocessor SRP and can be applied at partitioned scheduling, either with task-fix or job-fix priorities. Similar to D-PCP and M-PCP, the M-SRP distinguishes between local and global resources. The access to local resources is handled according to the SRP for uniprocessors. Besides, global resources use non-preemptive spin-locks with FIFO sorted wait-queues, where tasks add themselves to the end of the spin-queue, if a resource request is denied. Non-preemptive here means that both, the spinning and the critical section itself are non-preemptive.

Figure 3.21 shows an example schedule of the M-SRP, based on the same settings like the examples in Figure 3.19 or 3.20 respectively. As busy-wait and the critical section itself are non-preemptive, task T_1 gets activated at time 3 but gets blocked until task T_2 has released resource R_1 at time 6, although task T_1 has a higher priority than task T_2. Local resources that would be handled according to the uniprocessor SRP are not considered within this example. However, the schedule of the M-SRP ends one time unit after the M-PCP, but it causes fewer context switches like the D-PCP or the M-PCP [5]. A detailed blocking analysis of the M-SRP can be found in [73]. As a result of this

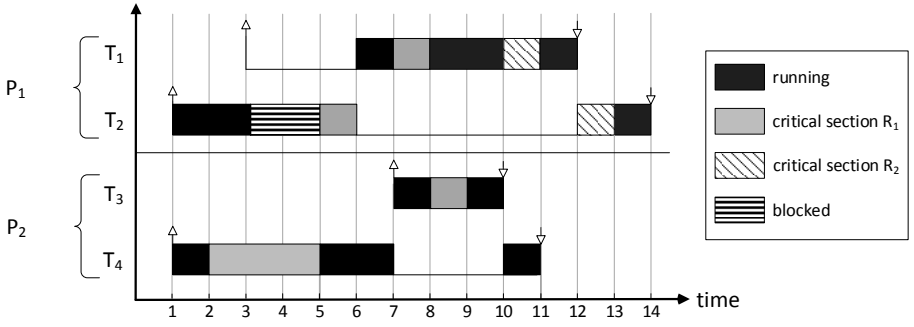

FIGURE 3.21: Example schedule with multiprocessor stack resource policy. Local resources are handled according to uniprocessor SRP, whereas critical section of global resources are carried out non-preemptively.

[5]Note, that the overheads caused by context switches are not considered in the depicted examples

analysis, the maximum spinning time is:

$$spin(T_i, R_l) = \sum_{\substack{k=1 \\ k \neq P_i}}^{m} \max \{CS_{k,l} \mid T_a \in \tau_k\} \tag{3.29}$$

Later, [83] presented a refinement of M-SRP: The parallel stack resource policy P-SRP generalizes M-SRP to the application of parallel (fix-priority) tasks. At the P-SRP, every tasks consists of a sequence of so called schedule segments. These schedule segments again consist of resource requirements which are either preemptive or non-preemptive. Requests to local preemptive resources are granted in priority order (queue-based), whereas all global resources are non-preemptive and handled in FIFO order. Local non-preemptive resources have a resource ceiling and are handled according to SRP rules, while local preemptive resources have a system ceiling and are handled according to SRP rules as well. Resources within a schedule segment are accessed simultaneously (parallel tasks).

b) Devi's Protocol [64] developed a non-preemptive FIFO spin-lock protocol for global EDF scheduling. Global scheduling means that jobs are not assigned fix to processors, and thus the maximum spinning duration can not be bounded like for the M-SRP. The protocol works similar to the M-SRP, with the exception that all resources are global which is caused by the applied scheduling algorithm. A maximum bound for spinning was derived in [64], which is:

$$spin(T_i, R_l) = (m-1) \cdot \max_{\substack{1 \leq k \leq n \\ k \neq i}} \{CS_{k,l}\} \tag{3.30}$$

c) Multiprocessor Resource Sharing Protocol MrsP Recently, [34] provided a further busy-wait based protocol for partitioned, task-fix priority scheduling, which allows nesting as well. It was inspired by the M-SRP from [72] and is called multiprocessor resource sharing protocol MrsP.

MrsP works similar to M-SRP with the exception that tasks that busy-wait can use their waiting time to execute resource requests of other tasks (on other processors) that are in front of the waiting task in the FIFO queue.

A short example for that is given in [34]: If a resource R_l is requested by two tasks T_i

and T_a ($Q_{i,l}$, $Q_{a,l}$) from two distinct processors then the critical sections associated with $Q_{i,l}$ and $Q_{a,l}$ can be executed by any of the two tasks. Assume that T_i accesses the resource first, but gets preempted while it owns the lock on R_l. Then, T_a is blocked, because T_i owns R_l. Under MrsP, it will first take over the execution for T_i, complete the critical section for T_i, and it will then execute its own critical section. When T_i gets resumed at a later point in time, it will find that its access to R_l has been completed.

d) Preemptive Waiting Synchronization Protocols A protocol with preemptive spinning is discussed in [53], where they developed a list-based preemptive waiting protocol for partitioned, task-fix priority scheduling. Preempted tasks have to be re-queued again at the end of the (priority ordered) queue (requests have to be re-issued). This re-issuing is accompanied by additional runtime overhead. [91] introduced ticket and list-based queue locks with preemptable spinning, which they named scheduler-conscious locks. It requires additional OS support to reliably recognize preempted tasks and is designed for partitioned, task-fix priority scheduling.

The preemptive waiting protocol from [130] uses interrupts to preempt spinning tasks. This approach was designed for partitioned, task-fix priority scheduling, and does not allow nesting of resource accesses. Within Takadas protocol, skipped tasks continue waiting when they are resumed. However, a blocking time analysis is not possible because there might be multiple tasks skipped or resumed unbounded times. Further, the principle of helping was presented in [131] where tasks execute critical sections on behalf of other, waiting tasks. Unfortunately this helping principle does not fit the task-model of many systems (and of this thesis), but this leads to a first approach for synchronization protocols of quantum-based schedulers in [7]. Locking protocols for quantum-based schedulers are discussed in Section 3.3.4.4.

3.3.4.4 Further Protocols

In this section, we present locking protocols which can not be declared as either suspension-based nor busy-wait based. The first protocol, the flexible multiprocessor locking protocol FMLP uses both techniques of waiting. Further, the multiprocessor bandwidth inheritance protocol is server-based and uses either busy-waiting or its server may be

used by a lock holding task. Finally, a family of locking protocols of Pfair scheduling is presented.

a) Flexible Multiprocessor Locking Protocol FMLP As mentioned above, the flexible multiprocessor locking protocol FMLP from [26] uses both waiting techniques, busy-wait and suspension. All types of scheduling in terms of allocation and prioritization are supported.

The decision whether busy-wait or suspension is used is made off-line by the system designer. Two different types of resources are available, either *short* or *long* resources. The terms short and long refer to the length of the critical sections.

Further, nesting of resource requests is eligible as well, if the principle of resource groups is applied. That means that each contains either long or short resource requests which are protected by a common group lock. In [26] is described that two resources R_1 and R_2 are in the same group if there exists a task that issues a request Q_1 for R_1 that is contained within a second request Q_2 for R_2 and further, R_1 and R_2 are either both short or long resources.

The idea behind the distinction between short and long resources is that for long critical sections, too much processor time is waisted when busy-wait is used, while for short critical sections, the overhead for context switches exceeds busy-wait time.

For each resource, the system designer has to decide whether it is short or long. Short resource requests are handled by non-preemptive busy-wait, whereas tasks that issue long resource requests are suspended, if their request is not successful. In contrast to short resources, critical sections of long resource requests are preemptive. The number of suspensions for long resources is bounded by applying priority inheritance. That means that (long) resource holding jobs inherit the maximum priority of any higher-priority job blocked on a resource in the same resource group.

The example in Figure 3.22 shows a schedule of the FMLP for short resource requests. These short resource requests are performed non-preemptively that means in this example that tasks T_1 and T_3 are blocked although they have higher priorities than the running tasks T_2 and T_3. The schedule on processor P_1 is here equal to the example of the M-SRP, but note that the FMLP does not distinguish between local and global resources. It is assumed that all resources are global. This means that under FMLP,

all short resource requests are non-preemptive (spinning and critical section), whereas under the M-SRP, only global resources are handled like this. Figure 3.23 shows an

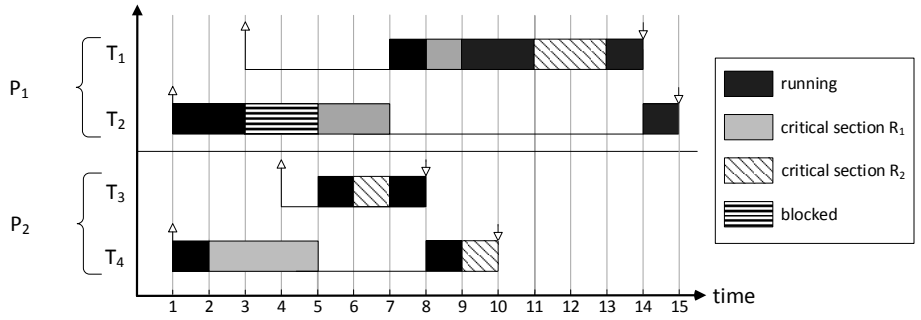

FIGURE 3.22: Example schedule with FMLP for short resource requests. Waiting tasks, for example task T_2 from time 3 to time 5, perform non-preemptive busy-wait.

example schedule for the FMLP for long resource requests. As described above, tasks that are blocked when requesting long resources are suspended, like it occurs at time 3 for task T_2. Followed by this, task T_1 is allowed to start execution until it gets blocked by the same resource. As requests are queued in FIFO order, task T_2 gets the resource R_1 at time 5 (and not task T_1 that has a higher priority than T_2), when task T_4 has released it. The blocking analysis in [26] comes to the conclusion that three different

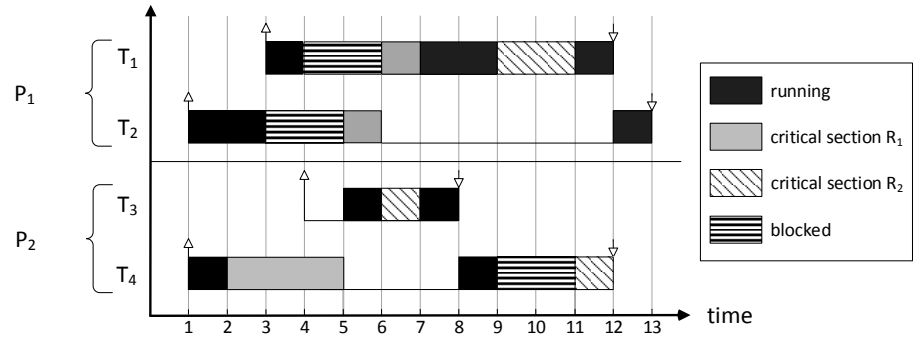

FIGURE 3.23: Example schedule with FMLP for long resources. Waiting tasks are suspended like task T_2 at time 3, so that other, not blocked tasks like task T_1 are able to execute meanwhile.

types of blocking may occur, namely busy-wait blocking, non-preemptive blocking, and direct blocking. The results of this analysis are presented below. Busy-wait blocking means the time a job $J_{i,j}$ spins for a resource that is held by another job $J_{a,b}$. The spinning time, $spin(T_i, Q)$ lies in between 0 for inner resource request up to Msum(m-1,S)

which is the summed spinning time of all short resource request of the actual resource group within the FIFO queue. Because nesting is allowed, busy-wait blocking BW of task T_i is:

$$BW(T_i) \leq \sum_{Q \in S(i)} spin(T_i, Q) \tag{3.31}$$

Where S(i) is the set of short requests issued by task T_i.

Next, non-preemptive blocking occurs if a job $T_{i,j}$ gets activated or resumed, but gets blocked by a higher priority job $J_{a,b}$ that holds a short resource and thus is non-preemptive. It can be calculated as:

$$\begin{aligned} NPB(T_i) \leq max\{np(J_{a,b}) : J_{a,b} \in B(T_i)\}+ \\ L(T_i) \cdot max\{np(T_{a,b}) : T_{a,b} \in A(T_i)\} \end{aligned} \tag{3.32}$$

Where $B(T_i)$ denotes the set of jobs of tasks other than T_i with a period that is larger than $p(T_i)$, $A(T_i)$ is the set of jobs of tasks other than T_i, and $L(T_i)$ is the number of (long) outermost resource requests that are issued by any job of task T_i.

Finally, direct blocking occurs, if a job $J_{i,j}$ is one of the m highest priority jobs, but gets blocked by another, lower priority job $J_{a,b}$ that holds a resource that is required by $J_{i,j}$. Direct blocking (for one single request) is:

$$\begin{aligned} db(J_{i,j}, Q) = \textstyle\sum_{T_a \in Z}(max\{np(J_{x,y}) : J_{x,y} \in A(T_a)\})+ \\ max\{ht(T_a, Q_a) : Q_a \in G(T_a)\}) \end{aligned} \tag{3.33}$$

Where Z is the set of tasks other than T_i that have issued a request for a resource in group g, and $G(T_a)$ denotes the set of (long) outermost resource requests by a job of T_a for a resource in group g. Because task T_i may issue more than one (long) outermost requests, the direct blocking for T_i is calculated as:

$$DB(T_i) \leq \sum_{Q \in L} db(J_{i,j}, Q) \tag{3.34}$$

Where L is the set of (long) outermost resource requests issued by an arbitrary job $J_{i,j}$ of task T_i.

b) Real-Time Nested Locking Protocol RNLP The real-time nested locking protocol was presented by [137] and supports partitioned, global and clustered scheduling

with job-fix priorities. It can be implemented either with busy-wait or suspension of blocked tasks and uses the k-exclusion resource model which has already been described in Section 3.3.1.1. Fine grained nesting is supported, where the resource constraints of the FMLP can be weakened. For the RNLP it is not longer necessary to combine resource requests into fixed groups statically, but only a partial order on resource acquisitions is required to ensure deadlock freedom.

Basically, the RNLP combines two different mechanisms. On the one hand, a token lock restricts the number of jobs within a critical section. On the other hand, if a job has acquired a token successfully, it is allowed to compete for a shared resource under the rules of an underlying *request satisfaction mechanism* RSM. The RNLP is not restricted to a certain token mechanism or RSM, different types of both techniques may be combined. These different types of mechanisms are described in detail in [137].

c) Multiprocessor Bandwidth Inheritance Protocol M-BWI The multiprocessor bandwidth inheritance protocol M-BWI was presented by [68] and is built for real-time open systems where tasks can be removed or added during runtime. It uses a server based approach, and it is an extension of the bandwidth inheritance protocol in Section 3.3.3.6. The M-BWI allows tasks that are holding a resource to use the time-budget from other tasks that are blocked while waiting for the same resource. This is possible because lock holding tasks are allowed to migrate between the different servers. A task that waits to access a resource either busy-waits (using its servers budget) or its server may be used by a lock holding task. If more than one task is waiting for a resource, access is granted in FIFO order.

d) Locking Protocols under Pfair scheduling Holman and Anderson presented different approaches to handle locks within Pfair scheduling in [84]. The first protocol makes quanta non-preemtable if critical sections can be kept within one quantum. This is a similar approach to Devi's protocol [64], but for quantum based schedulers. Further, for short critical sections which may exceed the length of one quantum, two different types of protocols are presented in [84]. On the one hand, zone-based protocols are offered, where quanta are checked if they are still within the allowed time-zone and thus are able to meet their pseudo-deadline. At the end of a quantum, there exists a so called *blocking zone*, in which tasks are not allowed to enter critical sections, otherwise

deadlines may be violated.

The second type of protocols uses skipping of requests, if requesting tasks are inactive. For bounding the skipped requests, different types of inheritance are possible use: *Rate inheritance*, *deadline inheritance*, *allocation inheritance* and *weight inheritance*.

Rate inheritance extents the concept of priority inheritance. A lock holding task inherits the highest task weight of the tasks that it blocks if it is higher than the own weight.

Under deadline inheritance, a lock holding task swaps its identity with that blocked task that has the highest priority at each time instant.

Allocation inheritance is a very close concept to bandwidth inheritance [96]. It improves deadline inheritance because it avoids unnecessary swapping overhead. The quanta of all blocked tasks can be redirected to the lock-holding task for the duration of its critical section. That means that multiple sets of (complete) scheduling parameters may be mapped to the same task at a given instance.

Finally, at weight inheritance the weight of each blocked task is added to the resource-holding task.

For tasks with long critical sections, Holman and Anderson referred to server-based approaches similar to that of Section 3.3.4.4.

Chapter 4

Contribution

This chapter provides the contribution of this thesis. First, its focus is discussed, followed by the the presentation of a global, dynamic priority scheduler for complex embedded systems, as well as two different multiprocessor real-time locking protocols.

4.1 Focus of Contribution

The objective of the contribution of this thesis is to optimize the temporal robustness and the efficiency of embedded multiprocessor real-time systems. The term temporal robustness refers to the goal that the central real-time requirement, namely to keep all deadlines of every task within a task set, are fulfilled despite the presence of disturbance like synchronization or system state transitions. For a more detailed definition of robustness, interested readers are referred to [59, 93, 19]. Further, efficiency means that the overhead that is spent in order to fulfill these requirements is minimized.

Basically, there are two main topics that have significant effect on robustness and efficiency of embedded multiprocessor real-time systems, namely scheduling and synchronization. Like discussed in Chapter 3, there already exist optimal scheduling algorithms for multiprocessor real-time systems. Nevertheless, there is still a gap to close between scheduling theory and practical systems. The first gap is that most of these practical systems have more complex requirements than theoretical ones. For instance, there is a need for system transitions that occur during runtime. Secondly, there may occur more than one activation pattern within one system. Periodic, sporadic and angle-triggered

activations may be combined.

Further, if practical systems are assumed, for example in the automotive powertrain domain, the approach of multiprocessors has just started and up to now, only partitioned, task-fix priority scheduling is applied. This scheduling scheme is accompanied with a row of drawbacks that are discussed later in Section 4.2.

Furthermore, all the locking protocols discussed in Section 3.3 are accompanied with specific drawbacks. For instance, many of the provided protocols require to apply a certain kind of task allocation or priorities or do not fit to the task model defined in Chapter 2. Furthermore, most of them do not support nesting. The only protocols that do not have such constraints are either difficult to implement (see the OMLP in Section 3.3.4.2, or the RNLP in Section 3.3.4.4) or lead to many priority inversions and unnecessary blocking times (see the FMLP in 3.3.4.4). However, there exists no locking protocol that outperforms all other protocols in every application, the performance of the best protocol depends mainly on the underlying hardware architecture and further on the workload of the system (see for example [138]).

The sum up of the thesis' contributions can be described in the thesis objectives. The first one is to create a scheduler that fulfills the following objectives:

- 1a) Increase the robustness of the system.
- 1b) Consider different types of activation patterns (see Section 2.6.2).
- 1c) Consider and handle system state transitions.
- 1d) Consider and handle core affinities of tasks.
- 1e) Allow different types of disruption (see Section 3.2.2).
- 1f) Provide an efficient synchronization mechanism across the cores.
- 1g) Keep the existing OS as basis (implementation efficiency).
- 1h) Allow the presence of parallel and chained tasks.

A detailed description of the described objectives and how they are met is given in the following sections.

Further, synchronization mechanisms shall be developed that fulfill the following objectives:

- 2a) Support all types of allocation and prioritization (see Section 3.2.2).
- 2b) Support nesting of resource requests.
- 2c) Minimize the blocking overhead (compared to existing mechanisms).

These objectives are discussed in Section 4.3.

Based on the gaps described above, we present an architecture concept of a global scheduler with job-fix priority scheduling, as well as two new locking protocols for synchronization and finally performance metrics to evaluate the robustness and efficiency of task sets via event-based simulation.

4.2 A Global, (Job-Level) Dynamic-Priority Scheduler for Complex Embedded Real-Time Systems

Up until now, practical multiprocessor systems in the automotive powertrain domain only use partitioned, task-fix priority scheduling approaches. This solution involves with several disadvantages. The first drawback of the partitioned task allocation approach is that tasks have to be assigned to cores at build-time and these partitions can not be changed during run time. Because of that it is not possible to reach fully the optimization goal, which is an optimal load balance regarding all cores. Different reasons lead to this optimization goal. First, if the CPU load can be balanced dynamically during runtime, one can avoid cases where one core is over-utilized and thus deadlines of one or more tasks are violated while another core remains idle. Further, if the load is well-balanced and there is free capacity left on all cores, the processing speed of the controller can be reduced in order to safe energy and reduce heat dissipation.[1] Third, if the load is not well-balanced across the available cores, those cores that have to process a higher load may suffer from aging earlier than the other cores. This leads to a shorter lifetime of the whole system. This problem will increase in future, when additional cores are available and thus bin-packing gets more challenging.

Further, if task-fix priorities are applied, the multiprocessor is not able to react on changing loads during runtime. This strengthens the load-balance problem additionally, and the task set that has to be performed by the controller might suffer from deadline violations if too many tasks are activated on one core simultaneously, even if other cores are idle at the same time.

Based on these drawbacks, one might conclude that global, fully-dynamic priority scheduling should be applied, as there exist two optimal algorithms (see Section 3.2.5.2). Unfortunately, these algorithms are accompanied by different inconveniences as well. Because priorities of tasks change fully dynamically, they have to be recalculated (for every task) every time the scheduler is called. As an example, under Pfair scheduling, the schedule has to be recalculated after each quantum. This leads to a high scheduling overhead. Further, followed by the rapid changes of priority, there occur many context switches and migrations when the schedules are recalculated.

Recently performed case studies confirm the assumptions that quantum based scheduler

[1] Note that energy consumption rises quadratically to the clock frequency of the processor.

are accompanied by significant overheads, for example the ones in [29, 31]. However, we further assume the practical requirement that the scheduling approach should be based on existing hardware, operating system and software, so that as few changes as possible have to be made. Further requirements are, that the scheduler has to be efficient (minimized scheduling overhead), it should be able to handle state transitions and data consistency has to be kept. Tasks that have core affinities have to be considered as well as chained tasks where the activation of a job depends on the termination of another job. The main parts of these contributions are in process to be protected by a patent (application number DE-102015218431.5) [77]. The next sections describe a concept of a scheduler that is able to fulfill all these requirements and provides a trade-off between the advantages of partitioned, task-fix priority scheduling on the one hand, and fully dynamic priority scheduling on the other side.

4.2.1 Starting Situation

One major requirement to the new scheduler is, that it should be based on an existing multiprocessor system. This is based on a shared memory multiprocessor that contains three heterogeneous cores (two identical cores, and one core with a lower processing speed but additional safety features). An overview, containing the memory model is depicted in Figure 4.1.

Each core has its own core local RAM memory, and additionally all cores have access to a global system RAM (via crossbar). Further, remote access from a core to the core-local RAM of another core is provided by the multiprocessor. This accesses are issued via crossbar as well. This memory architecture is important when shared resources are used, which is described later in Section 4.2.8.2.

Nevertheless, this concept should be able to be used for systems with n either symmetric, uniform heterogeneous or unrelated, heterogeneous multiprocessors.

Further, we assume an operating system for partitioned, task-fix priority scheduling, where one instance of the scheduler exists on every core. Figure 4.2 shows the task state model of such an instance. Initially, each task remains in the state *suspended*. When an activation event for a task T_i occurs, an instance of this task (job $J_{i,j}$) gets *activated* and becomes *ready*. Further, if job $J_{i,j}$ is selected by the scheduler for execution, it becomes dispatched and thus *running*, that means that $J_{i,j}$ is processed on the corresponding

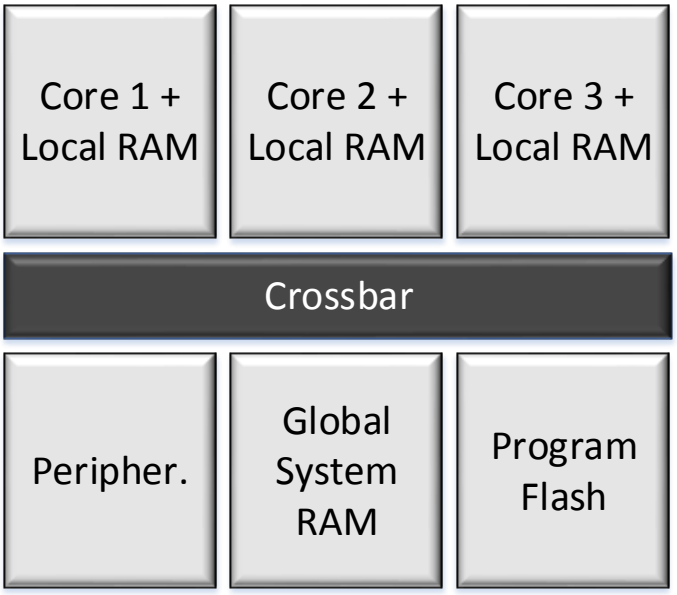

FIGURE 4.1: Multiprocessor architecture including memory model. Every core has its own local RAM, program flash and the global system RAM and peripherals are accessed via crossbar.

core. If another job $J_{a,b}$ is dispatched meanwhile, job $J_{i,j}$ gets preempted and becomes *ready* again. Finally, when the execution of job $J_{i,j}$ is finished, it gets terminated and task T_i becomes suspended again.

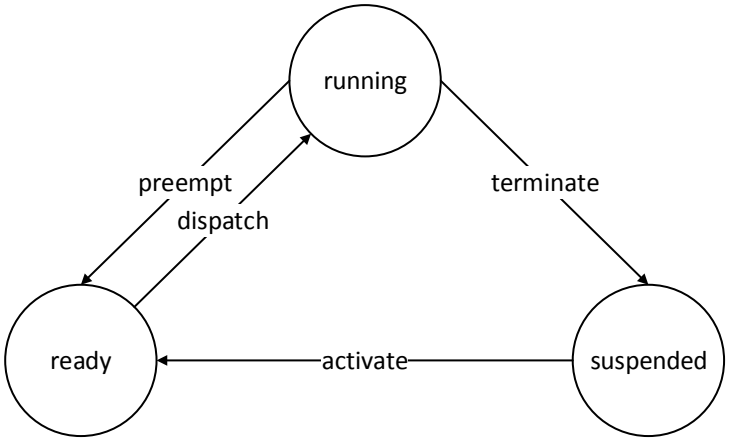

FIGURE 4.2: Basic task state model of the underlying task-fix priority scheduler that exists on every core of the basic system.

We assume systems where instances of tasks (jobs) may be issued by different activation patterns, namely time triggered, event triggered and angle-triggered ones. Additionally, some tasks have core affinities. That means that jobs of these tasks have to be processed

on a specific core (with suitable safety features).

During runtime, system state transitions may occur, as if the ignition key of the vehicle is turned on (engine off to engine on). The purpose and processing of state transitions is described later in Section 4.2.9. Finally, we assume a buffering system for synchronization of dependent tasks, which is discussed later in detail as well (see Section 4.2.8.2 and following).

4.2.2 Choice of Scheduling Algorithm

The first step of the scheduler concept is to chose, what kind of task allocation should be applied. We have already discussed the disadvantages of the partitioned allocation scheme. A further alternative, clustered scheduling, would not be convenient because the assumed hardware architecture contains only three cores. Therefore, clustered scheduling would be a future approach for hardware architectures containing additional cores. Because of that, we focus on a scheduler with global allocation. The main reason for this decision is that a global allocation provides the opportunity that the scheduler is able to balance the load dynamically during runtime and thus the robustness of the processed task set arises.

Further, to face the challenge to optimize the robustness of the task sets, it is required that the algorithm is able to react on dynamic changes of the system. As a consequence, task-fix priority scheduling algorithms would not be an adequate solution.

The known optimal multiprocessor scheduling algorithms deal with fully dynamic priorities, but they are accompanied by the disadvantage that frequently re-calculating the schedule and the amount of context switches and migrations lead to significant overhead [29, 31]. Therefore, these systems become insufficient in practical approaches.

Other scheduling algorithms with fully dynamic prioritization, for example the LLF [110] (see also Section 3.2.3.2) seem to fit best on the requirements of this scheduler. Unfortunately they go along with the drawback that priorities are changing continuously during runtime. Thus, the scheduler has to re-calculate the schedules at every clock cycle of the multiprocessor (for each core) which means additional runtime overhead for the scheduler.

Based on this, we came to the conclusion that a job-fix priority scheduler would overcome these disadvantages. Job-level priority scheduler are able to react dynamically

dependent on the behavior of the system. Further, priorities have to be calculated only once (when a job gets activated). Since the earliest deadline first algorithm EDF is known as an optimal scheduler for uniprocessor systems, we decided to use it for our approach.

Nevertheless, the concept should be open in a way that the EDF scheduling algorithm can be replaced by any other real-time scheduling algorithm. Only the global task allocation scheme is obligatory. However, the following sections describe the scheduler architecture in general and more detailed, based on the assumption that the global EDF scheduling algorithm is used.

4.2.3 General Architecture of the Scheduler

The proposed global scheduler is developed to run physically on one predefined core of the multiprocessor. This approach minimizes synchronization overhead while calculating the schedules (running the scheduler) and dispatch tasks.

Like in the already existing multiprocessor system there is an instance of a task-fix priority scheduler on each core. The main advantage of this approach is that one can make use of the available features for task-handling, e.g. the handling of context switches or the task termination function. However, by adding a global EDF plug-in, one can anyway take advantage of the global EDF behavior.

The general architectural structure of our global EDF scheduler is depicted in Figure 4.3.

Under global scheduling, the assignment of tasks to core is only temporarily valid for one instance (job) of a task. The scheduler is triggered whenever a job gets either activated or terminated. For simplification, we first assume that non-preemptive scheduling is applied. When different kinds of preemptions (cooperative or fully preemptive disruption) are enabled, further modules have to be added which are described in chapter 4.2.6.

The global EDF plug-in calculates the schedule for all activate jobs (which job is allowed to execute on which core) according to the global EDF rules. When the EDF plug-in has decided that a certain job has to run on a specific core, this job is handed over to the real-time operating system RTOS (partitioned scheduler) on the corresponding core. The next sections describe the architecture of the scheduler plug-in more in detail.

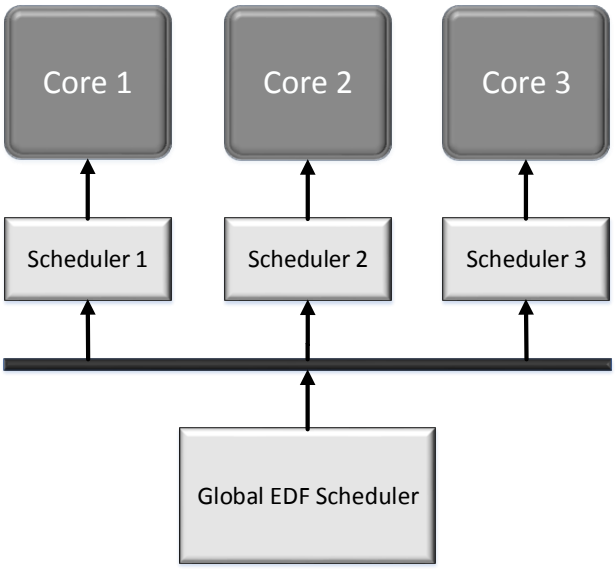

FIGURE 4.3: General structure of global EDF and partitioned (task-fix priority) schedulers for application on the assumed hardware architecture.

4.2.4 Detailed Architecture

The developed scheduler can be seen as an extension of a former approach for uniprocessor systems: Diederichs et al. developed an EDF scheduler plug-in for OSEK/VDX [116] in 2008 [66]. This approach is protected by the patent DE-102007042999 [76]. The OSEK/VDX fits to our model of partitioned, task-fix priority schedulers that are available on every core. They extended the basic task state model from Figure 4.2 by a further state called *delayed*, where jobs are sorted in a list by absolute deadline. Only the head job of this *delayed* list gets scheduled and thus dispatched on the core. The introduction of this additional state makes EDF behavior possible for uniprocessor case. The extended state model of the adapted approach from Diederichs is depicted in Figure 4.4.

Different to the basic task states, jobs that are in state *suspended* get *edfActivated* instead of *activated* and enter the state *delayed* if a trigger event for that job occurs. If a job gets activated it is inserted into a task list which is sorted by absolute deadline (ascending). Therefore, the job with the earliest absolute deadline becomes head of the list. Once a job is head of the list it gets activated and so it enters state *ready*.

The behavior of the EDF module (state *delayed*) at job activation from Diederichs et. al [66] is shown in Figure 4.5 more in detail. EDF behavior can be assured if and only

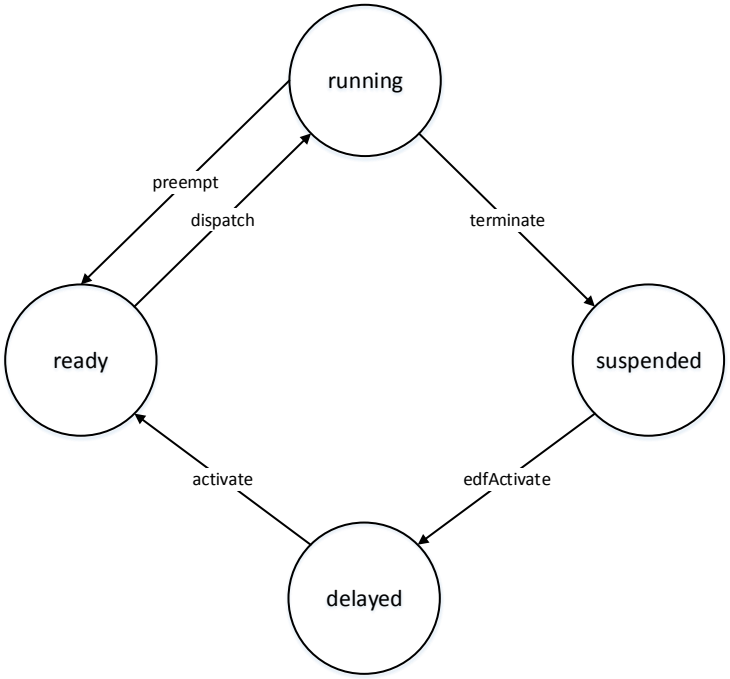

FIGURE 4.4: State model extended with the EDF plug-in for single core processors from Diederichs et al.[66]. The state *delayed* is introduced as a supplement of the basic task model in Figure 4.2 which contains the states *suspended*, *ready* and *running*.

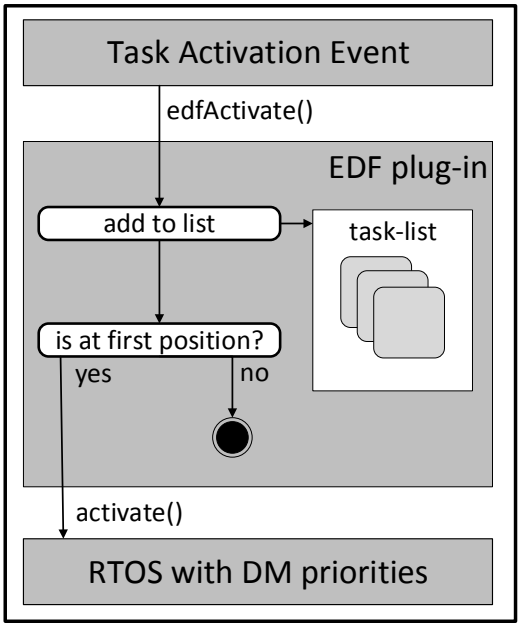

FIGURE 4.5: Singlecore EDF module behavior in case of task activation [66]. Tasks are sorted in a task list and the head of the list gets activated.

if the head of the list is the only job which gets activated and passes through into state *ready*.

Figure 4.6 shows the detailed behavior of the EDF module at job termination. This is also an important part to assure a correct EDF scheduling scheme. If a job gets terminated (back to state *suspended*) because it has finished its execution, it is removed from the list. After this, the new head of the list has to be *activated* (if it is in state *delayed*), so that the core does not remain idle.[2]

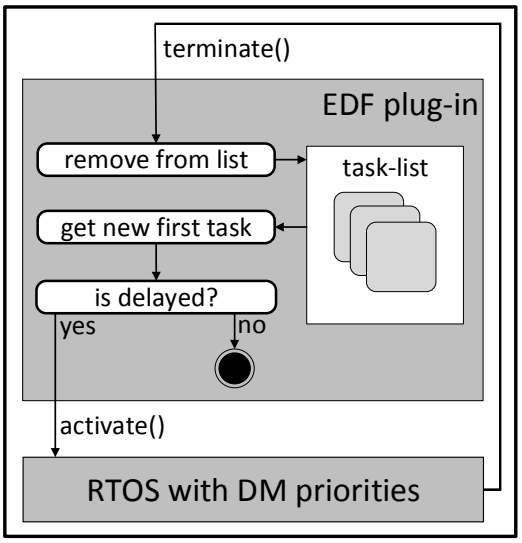

FIGURE 4.6: Singlecore EDF module behavior in case of task termination [66]. When a running job is terminated, the head of the task list gets activated if it is in state *delayed*.

Inspired by the contribution of Diederichs, we now adapt the EDF scheduling module to multiprocessor systems, which allows global and job-level dynamic allocation of tasks instead of the existing partitioned approach. If multiple cores have to be managed by one scheduler (global scheduling), the complexity of the systems rises. Note that one of the defined requirements to the system is that an instance of the existing (single-core) RTOS runs on every core. Figure 4.7 shows the states for the example of three cores that results from this requirements. Our extension contains the states *ready* and *running* m times, where m is the number of cores to be managed. However, the states *suspended* and *edfActivated* have only single occurrence and are globally available.

However, the global EDF scheduler has two main tasks. On the one hand, the m earliest deadline jobs have to be activated. On the other hand, it has to make a decision on which core a task has to be executed. The general architecture of the global EDF plug-in is drawn in Figure 4.3.

[2]Note that if a job is head of the list but not in state *delayed*, it has already been activated before and is either in state *running* or rather in state *ready*.

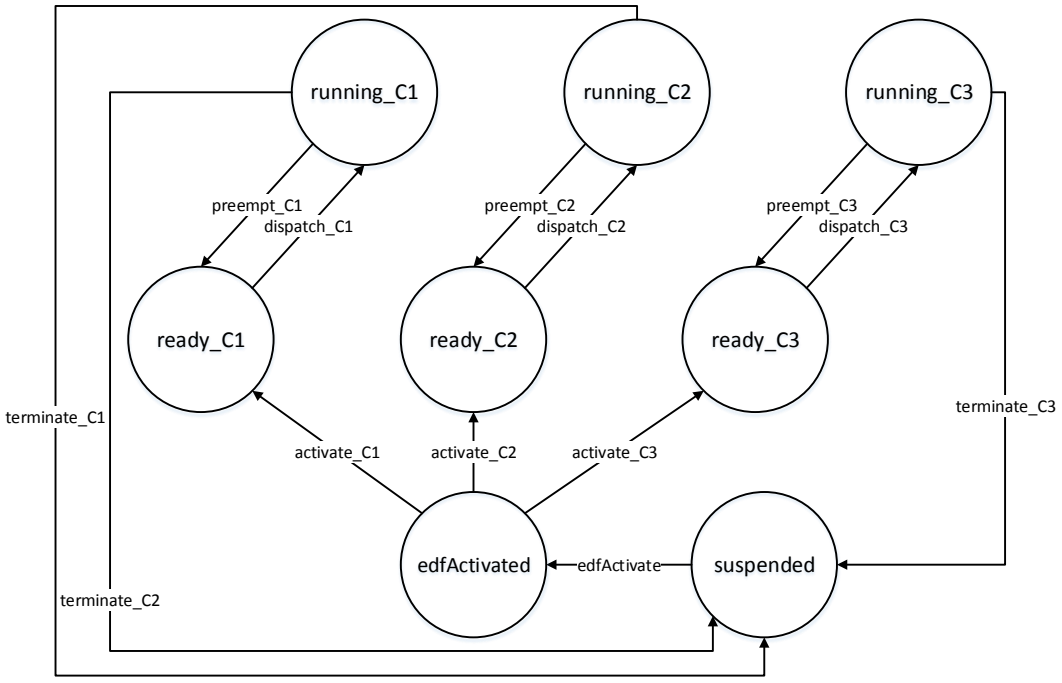

FIGURE 4.7: Task state model of the global EDF scheduler plug-in with m = 3 in this example. One instance of the states ready and running exists m times.

The global EDF approach is divided into different layers. Figure 4.8 gives an overview of this structure. In addition to the existing partitioned task-fix priority approach, two

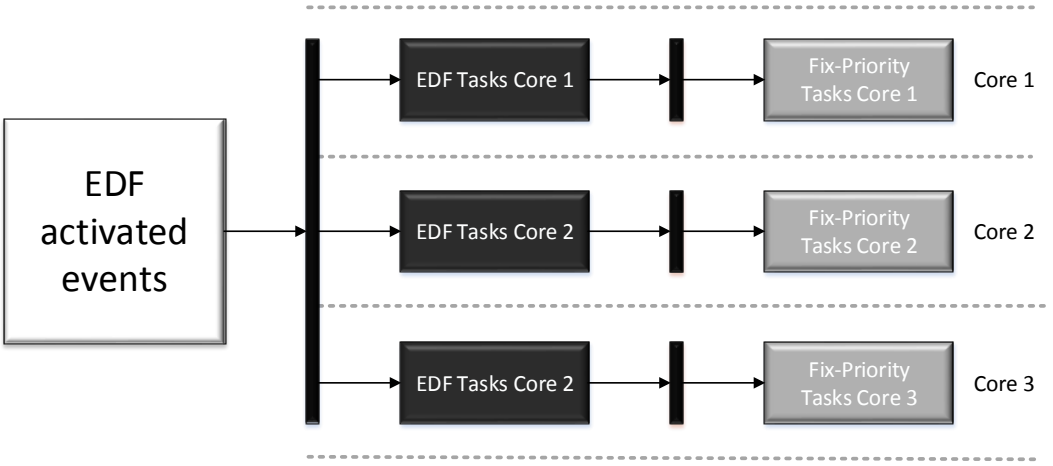

FIGURE 4.8: General structure of the different layers of the global EDF scheduler for m = 3.

additional layers are necessary for applying global EDF scheduling. The different layers are located in the modules *edfActivated* and *ready_Cx* respectively.

A global event list *EL* represents all EDF-activated jobs, whereas core specific lists (core lists) *CL* for each core represent jobs which have to be activated on a certain core. Note that for an implementation it would be favorable to manage the core specific list globally and to send only the activations to the correlated RTOS on the dedicated core. By acting this way, consistency of the global and core specific lists can be guaranteed. In between these two layers the decision on which core a job has to be activated, is made. The following sections discuss the different parts of the global EDF plug-in in detail.

4.2.4.1 EDF activate

In the global event layer, all EDF-activated tasks are represented by an entry within a global event list. Instances of activated tasks are called *schedule event* within this layer. Schedule events contain information about that piece of code which has to be executed (which functions have to be called).

At every *EDF-Activate* event of a task T_i (activation trigger for a job), the absolute deadline is calculated by summing up the EDF-Activation time-stamp ($a_{i,j}$) of the schedule event and the relative deadline (d_i) of job $J_{i,j}$: $D_{i,j} = a_{i,j} + d_i$

After that, the schedule event is inserted into the global system event list (block *EDF activated events* in Figure 4.8). This global event list is sorted by increasing absolute deadlines of schedule events. The gobal EDF module behavior can be seen in Figure 4.9. After the new activated schedule event is inserted into the global list, the *edfSchedule* function is called.

4.2.4.2 EDF-Schedule

Any time a new job is either activated or terminated respectively, the function for global EDF scheduling is processed. The behavior of the EDF-schedule module is depicted in Figure 4.10. First, the algorithm iterates through all local core lists (which are sorted by absolute deadline as well) and picks up its head elements that have the latest remaining absolute deadline each. If one or more local core lists are empty, the corresponding EDF core lists are preferred for inserting the head of the global schedule event list (if more than one core is idle, the selection is made in round robin manner).

Further, the absolute deadline of the head of the selected core list (if not empty) is

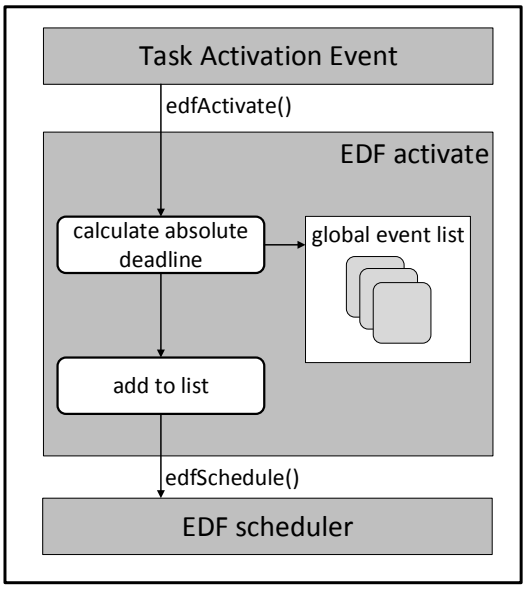

FIGURE 4.9: Global EDF activate module behavior at job activation. Activated jobs are inserted as an event into a global event list and sorted by absolute deadline. After that, the module *edfSchedule* is called.

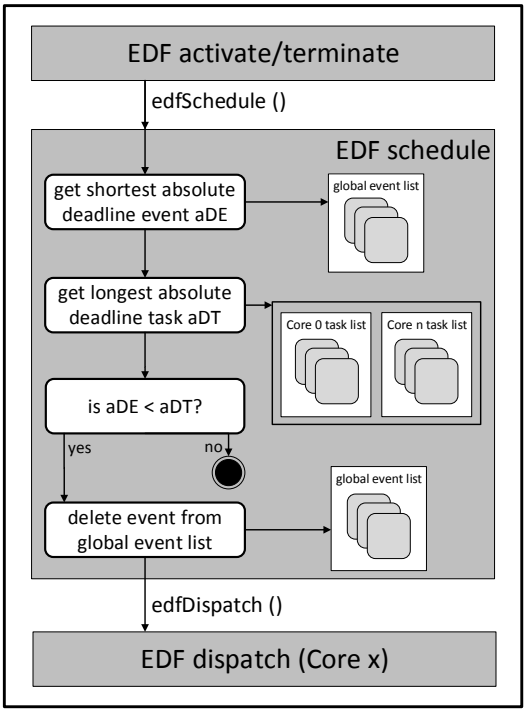

FIGURE 4.10: *EDF-schedule* module that selects a core and distributes selected, activated jobs (jobs that have one of the m earliest deadlines of the global event list) onto the dedicated core according to the global EDF rules.

compared with the absolute deadline of the head of the global schedule event list. If the absolute deadline of the global schedule event list is later than the absolute deadline of the selected head element of the core list, the algorithm stops because the m earliest

deadline jobs are already *running* on the available cores. Moreover, if the absolute deadline of the head inside the global schedule event list is earlier than the absolute deadline of the head element inside the selected local core list, the job from the global event list is inserted into the local core list and thus becomes head of this list. Then, the EDF-dispatch function is called.

The algorithm repeats this proceeding until the actual head of the global schedule event list has a later absolute deadline than the head element of the core list which has the latest absolute deadline. If this is the case, the algorithm stops because the abort criterion (the m earliest deadline jobs are running) is met.

4.2.4.3 EDF-Dispatch

Whenever the head element of a core list is not actively executing (in state *running*), this job gets EDF-dispatched. That means that this job gets activated and turns into state *ready* on the RTOS of the corresponding core.

This activation is called every time after the EDF-schedule function was performed. The behavior of EDF-Dispatch is drawn in Figure 4.11.

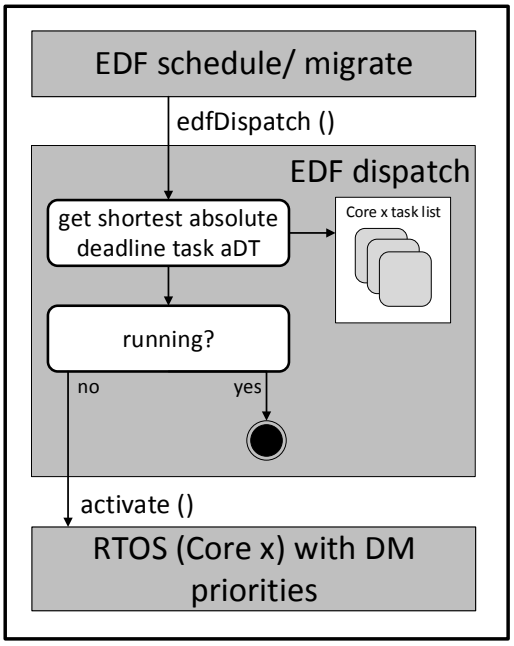

FIGURE 4.11: Global EDF module behavior for EDF-Dispatch where tasks are overhanded to RTOS state *ready*.

4.2.4.4 EDF-Terminate

Whenever a job has finished its execution it gets terminated. Now it is necessary to delete the job-entry from the EDF core list of the core it was executing on. Further, the RTOS task termination function is called in order to terminate the task properly. Finally, the module EDF-schedule (already described in Section 4.2.4.2) has to be called because the next (active) job may be ready to be executed on this core. The behavior of the EDF-terminate module is presented in Figure 4.12.

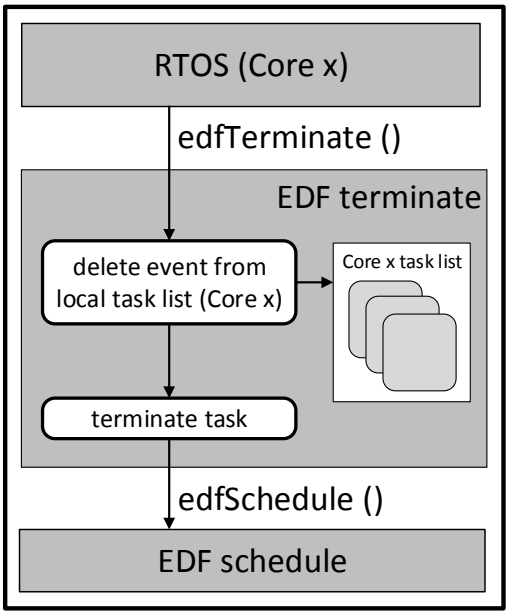

FIGURE 4.12: Behavior of the global EDF module in the case of task termination. The terminating job gets deleted from the corresponding core list, gets terminated and the EDF-Schedule module is called.

4.2.5 Verification of EDF Behavior

This section gives a verification of the correct behavior of the our scheduling approach regarding global EDF rules. We still assume that non-preemptive global EDF scheduling is applied.

First, we denote a task set τ to consist of a certain number of tasks $\tau = \{T_1...T_m\}$. A task T_i is a piece of software which has to be executed. A job $J_{i,j}$ is the j^{th} instance of a task T_i. When a global EDF scheduler is approached, a job is generally allowed to be executed on every of the m available cores of the system. A task contains several

parameters which represent its timing behavior. Like described in [66] a basic task consists of the following parameters:

$$T_i = (a_{i,j}, e_i, D_{i,j})$$

Where $a_{i,j}$ is the activation time of the j^{th} job of the i^{th} task, e_i is the execution time and $D_{i,j}$ is the absolute deadline of the j^{th} job. e_i and $a_{i,j}$ depend on the system behavior and scheduling schemes, while $D_{i,j}$ can be calculated as follows:

$$D_{i,j} = a_{i,j} + d_i \tag{4.1}$$

The relative deadline d_i of a task T_i has to be configured by the system architect. For further processing, we assume:

$$\forall(T_i \in \tau) = (a_{i,j}, d_i, D_{i,j}) \in \mathbb{R}^+ \tag{4.2}$$

For a more detailed description of the algorithm of global EDF scheduling over n cores, further task/job parameters are required:

$$T_i = (a_{i,j}, e_i, D_{i,j}, prio_i, st_i, lpEL_{i,j}, lpCL_{i,j})$$

These parameters are the (static) task priority $prio_i$, the task state st_i, the list position of job $J_{i,j}$ of task T_i in the global event list $lpEL_{i,j}$ and finally the list position of $J_{i,j}$ in (one of) the n core lists $lpCL_{i,j}$.

A correct real-time behavior of the system is only possible if the absolute deadline of a job $J_{i,j}$ is always greater than its activation time (otherwise its deadline is violated before $J_{i,j}$ has started its execution):

$$\forall(J_{i,j} \in \tau) : D_{i,j} > a_{i,j} \tag{4.3}$$

We continue with the assumption that the jobs within the global event list EL are sorted in ascending order by their absolute deadlines:

$$\forall(J_{i,j}, J_{a,b} \in \tau) : (D_{i,j} > D_{a,b}) \Longrightarrow (lpEL_{i,j} > lpEL_{a,b}) \tag{4.4}$$

All the local EDF core lists CL are required to be sorted in ascending order according to their absolute deadline as well:

$$\forall(J_{i,j}, J_{a,b} \in \tau) : (d_{i,j} > d_{a,b}) \Longrightarrow (lpCL_{i,j} > lpCL_{a,b}) \tag{4.5}$$

To guarantee that the task set is scheduled in EDF manner, it is required that the m jobs with the shortest absolute deadlines are executed. To reach this, first it is necessary that Equation 4.4 is fulfilled. As a result, the global event with the earliest absolute deadline is the head of the list.

Further, as soon as a job $J_{i,j}$ becomes either EDF-activated or terminated the schedule function has to be called and processed iteratively until one of the two abort criteria is fulfilled: Either, the global event list *EL* is empty which means that there is no job in state *ready* that has to be scheduled, or if Equation 4.6 is true.

$$D_{a,b}\{J_{a,b} : lpEL_{a,b} = 0\} > D_{x,y} \tag{4.6}$$

Where $D_{x,y}$ is calculated according to Equation 4.8, and $J_{a,b}$ is that job that is head of the global event list. The schedule function can be described as follows:

First, it has to be checked (for all m cores) if there is a core list CL_m on which no job is executing.

$$\exists\{CL : size(CL) = 0\}? \tag{4.7}$$

If this is the case, the job $J_{a,b}$ at the top of the global event list ($lpEL_{a,b} = 0$) is activated and thus inserted into the empty core list CL of core C_m. If more than one core list is empty, jobs are inserted into the core lists in round robin manner. In this case of course job $J_{a,b}$ will be the new head of the core list ($lpCL_{a,b} = 0$).

Moreover, if jobs are running on each core and thus every core list contains at least one job, the scheduler needs to get exactly that running job $J_{x,y}$ from its core list with the latest absolute deadline $D_{x,y}$ [3]:

$$\forall(J_{x,y} \in \{\tau : lpCL_{x,y} = 0\}) : getJ_{x,y}\{J_{x,y} : D_{x,y} = max(D_{x,y})\} \tag{4.8}$$

[3]Note that only the heads of the core lists are taken into consideration because these jobs are that ones which have to be preempted possibly (depending on the deadlines of running and waiting jobs.)

Now a comparison is made whether job $J_{a,b}$ at the top of the global event list ($lpEL_{a,b} = 0$) has an earlier deadline than the job with the job $J_{x,y}$ found by Equation 4.8. If $D_{a,b} < D_{x,y}$, job $J_{a,b}$ gets activated on that core where $J_{x,y}$ is running currently (and thus job $J_{x,y}$ has to be preempted). Furthermore, if $D_{a,b} > D_{x,y}$, the scheduler aborts because if Equation 4.8 is true and $D_{a,b} > D_{x,y}$, the m activated jobs with the earliest absolute deadlines are already running. Further, if the global event list is empty (there are no activated jobs pending for being scheduled), the algorithm is allowed to abort, too.

Once the schedule function has been finished (one of the two abort criteria is fulfilled), the *EDF-dispatch* function is called. The *EDF-schedule* module assures that the m jobs with the shortest absolute deadlines are correctly activated and distributed onto the core lists. In contrast, the purpose of the dispatch module is to preempt jobs and let other ones execute if necessary (according to EDF scheduling policy).

To reach a correct EDF behavior even with the present RTOS on every core although this is based on task-fix priority scheduling. The priorities of the tasks have to be assigned in deadline monotonic DM scheme[4]:

$$\forall(T_i, T_{,y} \in \tau) : (d_i > d_l) \Longrightarrow (prio_i < prio_l) \tag{4.9}$$

We assume that Equation 4.5 is fulfilled. The dispatch algorithm checks for each core list if its head job $\{J_{a,b} : lpCL_{a,b} = 0\}$ is currently activated on the RTOS. If this is not the case, the dispatcher activates job $J_{a,b}$, beyond that nothing has to be done. Contrary to this, if job $J_{a,b}$ gets activated on the RTOS on one of the m cores, it has to be guaranteed that it becomes running (it gets activated on the RTOS only if it is necessary according to the EDF rules). In order to realize this, the shortly activated job $J_{a,b}$ needs to have a higher priority than the currently running job $J_{x,y}$:

$$\forall(J_{a,b} \in CL) : (lpCL_{a,b} < lpCL_{x,y}) \Longrightarrow (prio_{a,b} > prio_{x,y}) \tag{4.10}$$

This equation 4.10 is fulfilled if both, Equation 4.9 and Equation 4.5 are fulfilled as well. Now it is guaranteed that a new activated job with shorter deadline $J_{a,b}$ will preempt the running job $J_{x,y}$ according to the scheduling rules in the RTOS and hence according

[4]In DM priority assignment jobs are sorted by relative deadline. The highest priority is assigned to the job with the shortest relative deadline.

to the global EDF rules.[5]

Finally, if a job $J_{i,j}$ terminates because it has finished its execution, it has to be deleted from the core list CL it was activated in and from the global event list as well.

4.2.6 Cooperative Disruption

Up to this point we followed the assumption that non-preemptive scheduling is applied. This approach is accompanied by the drawback that unnecessary deadline violations occur if early deadline tasks get activated while other tasks with late deadlines but high utilization are running.

However, to avoid this drawback, some kind of preemption has to be allowed. Since making a task fully preemptive causes significant overheads (save and migrate task context and processed data, handle many possible preemptions), we decide to apply a compromise between non-preemptive and fully preemptive disruption, namely cooperative disruption.

Generally, in case of cooperative disruption, migration does not cause high overheads for preempted jobs within the (assumed) existing system, as the user stack is cleaned up at preemption points. Further we assume a suitable method to buffer the data (the topic of buffering is discussed later in Section 4.2.8).

There, in case of task activation, EDF behavior can be guaranteed by the described modules above. Cooperative behavior is realized by the existing fix-priority schedulers on each core. However, the one scenario arises where EDF behavior would be violated: If a running job $J_{i,j}$ terminates and the concerning core remains idle after that (assuming that there is no waiting event in the global event list), there might exist other, preempted jobs, e. g. job $J_{a,b}$, that are not running on the other core lists currently (preempted means that $J_{a,b}$ had been running at a point of time in the past).

This means that there are jobs waiting for execution while a core is idle and thus is available for execution, which contradicts EDF behavior where the m (activated) earliest deadline jobs have to be executed.

A solution for that problem might be that in case that a job terminates, the opportunity

[5]Note, that a global EDF scheduler can work either in preemptive, cooperative and non-preemptive manner.

of job migration is offered. In the following we describe what additional modules are necessary for applying cooperative scheduling with migrations. First, after the terminated job $J_{i,j}$ is deleted from its corresponding core list, a new module called *migration-check* has to be called. The new module for task termination at cooperative scheduling is presented in Figure 4.13. The EDF migration-check module tests whether there exist

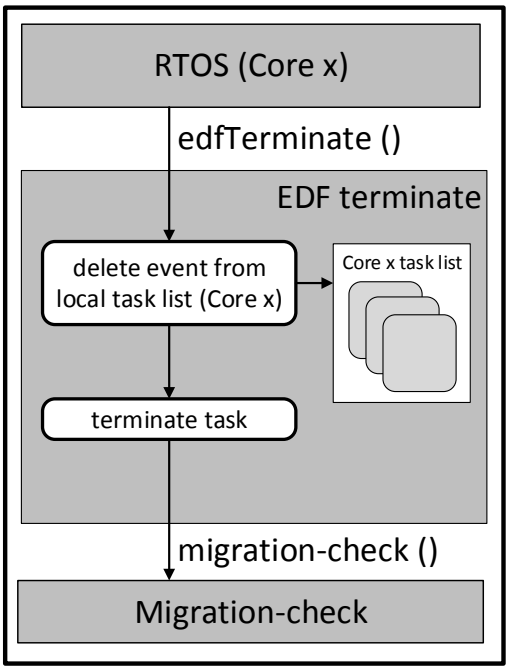

FIGURE 4.13: EDF terminate module for cooperative scheduling with possible job-migration. After the termination of a job, the migration-check module is called.

preempted jobs on any local core list. If not, EDF-schedule is called like in the non-preemptive approach. Otherwise, the earliest deadline preempted job $J_{a,b}$is compared with the head of the global event list, job $J_{k,l}$. If $J_{k,l}$ has a later deadline than $J_{a,b}$, the module EDF-migrate has to be called. In contrast, if $J_{k,l}$ has an earlier deadline, the EDF-schedule function is called. This processing is depicted in Figure 4.14. In the case that the result of the migration-check detects the need for a job migration, the EDF migration module is called. The migration module is drawn in Figure 4.15. We assume that core list x is that list, where the preempted job $J_{a,b}$ is contained. In contrast, core list y denotes that list that contains the jobs on the idle core where job $J_{i,j}$ was execution before termination (this list is empty at this moment). The EDF migrate module takes the preempted job $J_{a,b}$ and migrates it to core list y. After the migration step is finished, the EDF-dispatch module has to be called in order to activate the migrated job within the RTOS of core y.

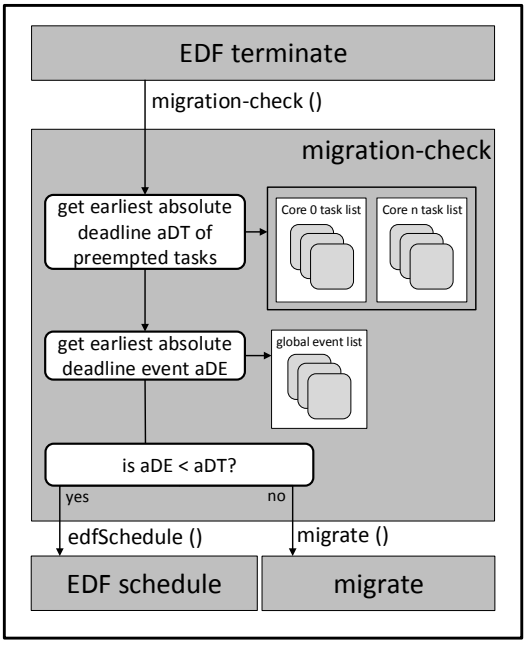

FIGURE 4.14: EDF-migration-check module. It checks whether a need for job-migration exists or not.

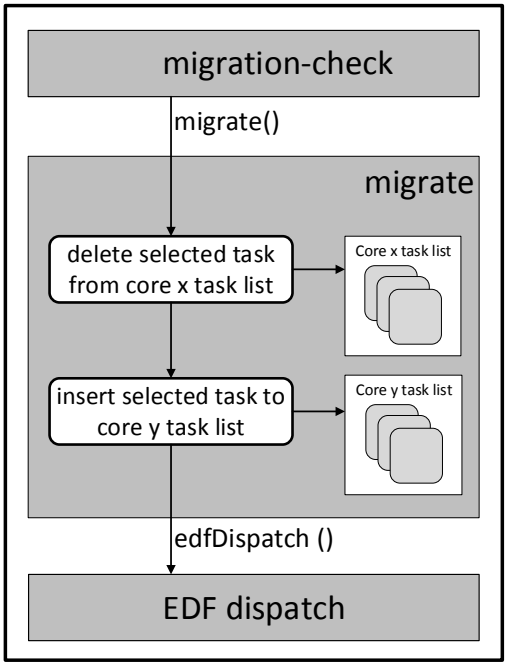

FIGURE 4.15: EDF migrate module for cooperative scheduling. Preempted jobs are removed from their core list and inserted in another core-list whose core remains idle.

By adding the modules migration check and migrate, EDF behavior can be guaranteed (the m earliest deadline jobs are running) for the case of either cooperative or preemptive scheduling. Nevertheless, recall that fully preemptive disruption (in contrast to cooperative disruption) causes overhead in terms of saving and migrating the task context when a task is preempted during execution.

4.2.7 Core Affinities

Some tasks within the task set may contain requirements regarding the core they have to be executed on. These requirements are caused for instance by safety-requirements according to ISO26262 combined with the provided architecture of the assumed processor, where certain cores offer different safety features[6]. Subsequently, some jobs have to be executed on a particular core only. All jobs of this task then have a *core affinity*.

If a job $J_{i,j}$ has such a core affinity, the modules EDF-activate and EDF-schedule can be skipped. $J_{i,j}$ is added to the corresponding local core list immediately. This can be done because the decision on which core it has to be executed has already been given by the core affinity. Nevertheless, this job is only executed in state *running* as soon as it has the earliest absolute deadline of all jobs that are queued in the local core list (by the module EDF-dispatch). Thus the occurrence of tasks with core affinities does not influence the system regarding EDF behavior (Tasks with earlier deadline that are activated meanwhile get a higher rank in the local core list or are assigned to another cores respectively).

4.2.8 Data Synchronization

The physical dependencies of an engine control system makes sharing of common resources across tasks necessary. We assume the following existing synchronization model: Semaphores are used to realize a reader/writer exclusion where multiple reading is allowed (counting semaphores), but multiple writing and also simultaneous reading and writing is prohibited. Read and write request are issued in FIFO manner.

However, additionally there might arise needs regarding data consistency and stability of produced and consumed data. A buffering mechanism assures that these specified

[6]for example the Infineon AURIX TC27x processor

needs can be guaranteed by the system. Such a mechanism is already in use in the basic multiprocessor approach that works with partitioned, task-fix priority scheduling. In the following section, the needs of data stability and consistency are pointed out. Further, different buffering concepts are introduced and a case study is carried out in order to make a decision about which concept should be applied.

4.2.8.1 Stability and Consistency Constraints

The buffering system is required to assure that the needs for data items to be either stable or consistent over a specific duration is fulfilled.[7] Using buffers to provide data stability and consistency has the advantage that the synchronization overhead can be reduced because synchronization is only necessary at the point in time where buffers are filled or written back.

This synchronization is performed by the use of global semaphores, each of them protecting a specified memory region. The main disadvantage of this approach is that tasks may work on old (buffered) data. Updates (during a buffering period) of a task are not immediately visible to other tasks. The focus of this work is only on the possible best usage of the existing buffering system in context of cooperative global EDF scheduling, optimizing the buffering mechanism as itself is out of scope of this work.

Figure 4.16 shows an example of where a data item (resource R_1 in this example) requires data stability. Runnable S1 (performed for example on Core 1) is reading resource R_1 two times. Between the first and the second read event, R_1 has to be stable. Assume that runnable S2 (executed on Core 2) updates resource R_1 between the two read events for R_1 from runnable S1. The need of stability for R_1 between both read events of runnable S1 would be violated (by the update of runnable S2). To avoid this, resource R_1 has to be buffered while runnable S1 is executed.

In contrast, the scenario in Figure 4.17 shows the need for buffering when data consistency is required for resources R_1 and R_2 while runnable S1 is running.

Runnable S1, which is executed on Core 1 here, requires data items R_1 and R_2 to be consistent between the read events. If Runnable S2 updates these variables meanwhile, the consistency can not be kept and the results of the calculations in runnable S1 would

[7] Note that only selected data items have to fulfill such requirements. There exist items that do not need any stability or consistency mechanism neither.

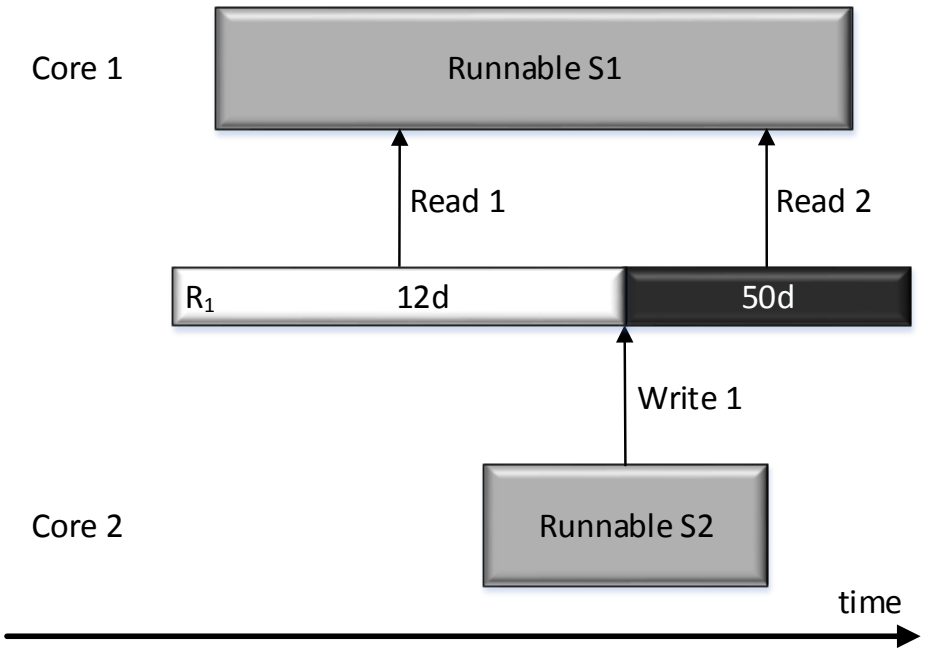

FIGURE 4.16: Example for a need of stability for data item R_1. If buffering is not applied, runnable S2 would update R_1 (from 12d to 50d) and consequently R_1 would not be stable anymore between the two read events of runnable S1.

be wrong. This may lead to unexpected behavior of the whole system. In order to avoid such a scenario, runnable S2 has to do a consistent write. For that, buffering and an atomic write back mechanism are required. This buffering mechanism is described in the next section.

4.2.8.2 Buffer Mechanism

The section above explained why applying a buffering mechanism is crucial to ensure data consistency and stability. Now it is presented how the mechanism itself works. An overview is drawn in Figure 4.18. Because we now describe the already existing mechanism, we assume partitioned scheduling within this section (prioritization does not matter within this part).

Buffers are allocated at the local core RAM where the owning task is assigned. When a task starts executing like Task T_1 in Figure 4.18, all data which are required to be buffered are copied from the global RAM into the buffer at the local core RAM where Task T_1 executes. Further, if there are data which do not contain any needs for buffering,

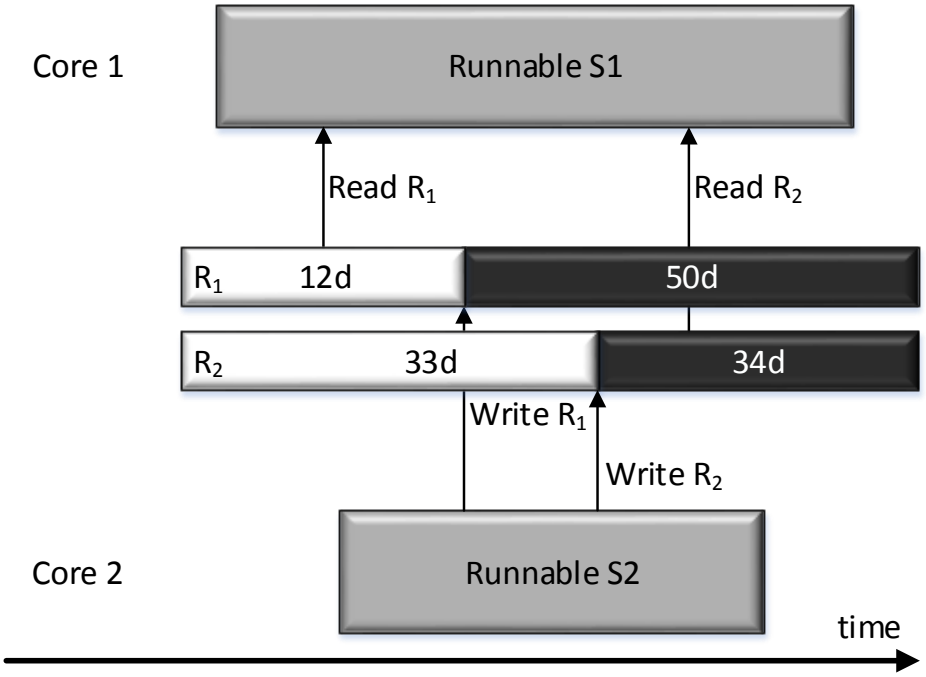

FIGURE 4.17: Example for a need of consistency for data items R_1 and R_2. If both items are not buffered, they would become inconsistent for runnable S1 between its two depicted read events.

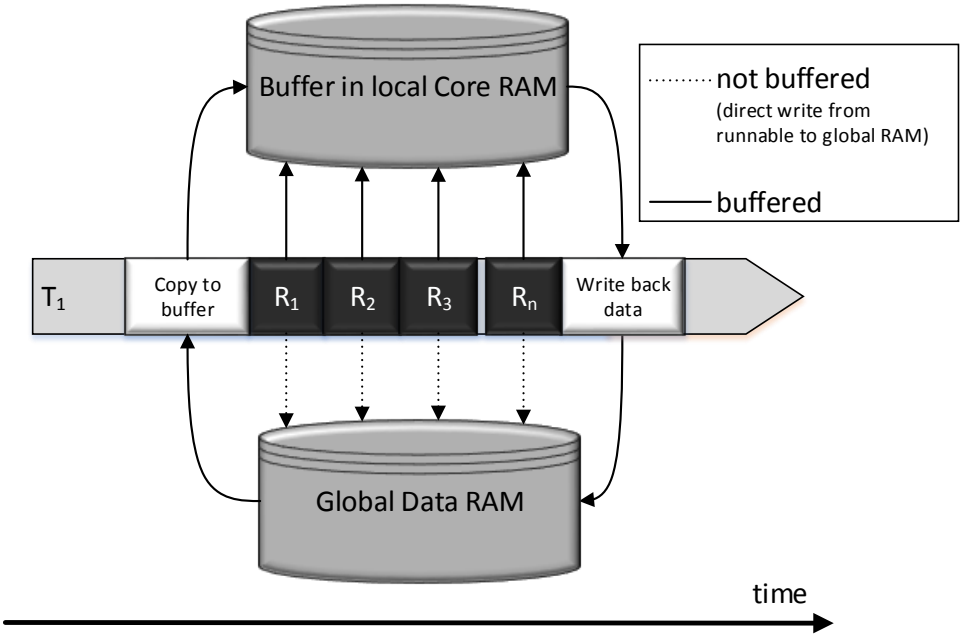

FIGURE 4.18: Schematic drawing of the buffering concept.

they can be written back directly from the accessing runnable to the according memory range in the global RAM without buffering.

Buffered data items are updated within their corresponding buffer at the local core RAM. After all runnables of the task have finished their execution, the data that are stored in buffers at the local core RAM are written back to the global RAM. This transaction is protected by a semaphore. When the write back has been finished, the updated data items become visible for all the other tasks. Runnables that have to be buffered follow these restrictions:

- All values which might be written shall be read in advance.
- All data have to be written consistently.
- The last writer wins.
- If elements of a structure will be updated by different writers, then only the update of one single element is allowed.
- Write back has to be atomic for data items bigger than 32 bit.

However, even if buffering is applied, semaphores are still required to assure that updates from the buffer to the global RAM are made consistently.

4.2.8.3 Buffer Concept For Partitioned Scheduling

Buffers are allocated on core local memory (memory space is reserved for buffers). An algorithm calculates required buffers at build time by analyzing stability and consistency needs, as well as the interference model of the task set. Tasks are able to access buffers fast and easily, because of the allocation (tasks never migrate, buffers are always stored at the core where the task is executing).

Nevertheless, this concept has to be reviewed when global scheduling is applied, where tasks may be executed on every core. This is done in the following sections.

4.2.8.4 Buffer Concept For Global Scheduling

There are two main differences regarding the buffering of data when task allocation changes from partitioned to global. First, all tasks may be interfered by any other task.

That means that more data items have to be buffered compared to partitioned scheduling. However, the algorithm that calculates the buffers can be easily adapted for this use case. Secondly, all tasks can be executed on every core and thus the corresponding task contents (e. g. buffers) cannot be assigned statically to one certain core as it is the case in the existing system with partitioned task allocation.

Two different approaches have to be discussed to find out how the handling of buffered items should be realized if tasks may be executed on different cores: Buffer migration on the one hand and remote access on the other hand. Both approaches are presented and reviewed below.

a) Buffer Migration The first option to manage buffering at global scheduling is to migrate buffers onto the core local RAM where their corresponding tasks are running. If data items have to be buffered across schedule sections, those sections might be executed on different cores (not necessarily, but possibly). If this is the case, the buffers of these data items are migrated across the local memories of the corresponding cores.

On the one hand, migrating the buffers offers the advantage that fast access to buffered items can be realized, because buffers are located close to the running task on the core local RAM. Unfortunately, this goes ahead with some drawbacks. First, additional runtime overhead is generated while buffers are migrated from one core local RAM to another one. Further, memory overhead arises as well (m times), because space has to be allocated (intended) for every buffer on every core.

b) Remote Access The alternative solution is to keep the buffer in the global main memory (or any core local RAM) and to access (read and write) this buffer from the core where the task is executed on remotely. This approach is accompanied by the advantages that no runtime overhead arises for migrating the buffers from one core local memory to another one. Further, memory space has to be allocated (intended) once only (either at core specific local RAM or at the global system RAM. The disadvantage of this approach is that run-time consuming accesses to remote buffers have to be performed (not only per buffer, but per access to any data item as well).

In order to be able to assess, whether migrating or remote access to buffers is more efficient, a data analysis was performed, where both options were reviewed. The results of this analysis is presented below.

4.2.8.5 Data Analysis

The goal of this analysis is to find out whether remote access or migrating is preferable when buffers have to be accessed from different cores. For this, an existing automotive powertrain multiprocessor project has been analyzed regarding the properties of the two different buffering approaches. For both examinations, for each task T_i we need to know its period P_i which denotes the number of activations per second. Time triggered tasks are activated in fixed periods, e. g. 10 ms or 1000 ms, while (engine) angle triggered task activation periods depend on the rotation speed of the engine. We assume an engine rotation speed of 6000 rpm which is at the upper level of engine systems. The following sections describe how the runtime and memory overhead for both buffering models is determined.

a) Buffer Migration If buffers have to be available across schedule sections and the schedule sections might be executed on a different core, the concerning buffers are migrated from the local RAM where the task was running before to the local RAM where the task has to continue after the migration. The worst case should be considered, so we assume that a buffer migration is required in between every schedule section.

In the following, we calculate the total runtime overhead of migrating buffers concerning the core utilization. The first step is to calculate the migrations per second. We assume that our system contains a task set τ containing an amount of n tasks T. We denote $bi\,{}^{n}_{m}$ to be the buffered items of schedule section m in task n. Further we need the period P_i of every task in τ (here in $\frac{1}{s}$). The migrations per second of a single task mps_i can be calculated as:

$$mps_i = [\sum_{j=1}^{m} bi^i_j]\ P_i \tag{4.11}$$

Further, *opm* is denoted to be the runtime overhead per migration. The migration overhead per second *ops* can now be calculated as:

$$ops = [\sum_{i=1}^{n} mps_i]\ opm. \tag{4.12}$$

The unit of *ops* is clock cycles per second. We assume to know the clock frequency f of the used processor, so we can now calculate the normed (total) overhead of buffer

migrations *nmo* (normalized to core utilization).

$$nmo = \frac{ops}{f} \tag{4.13}$$

The results of this analysis, as well as a benchmark to the remote access analysis is presented later in Section 4.2.8.5.

b) Remote Accesses The second method to provide stability and coherency of data in between schedule sections is to issue remote accesses to that memory where the buffers are stored. Thereby, the access has to get over the crossbar of the controller. This goes ahead with runtime overhead. We assume the kind of architecture in which it does not mind (in terms of runtime) if the buffers are allocated at a remote, local core memory or at the global system RAM (there is only a runtime difference to the own local core memory where the task is running). Considering the worst case, we assume that all data items have to be accessed remotely via the crossbar. We get the buffer accesses per task ba_i by analyzing the project data. Further, like in Section 4.2.8.5 the period P_i is denoted to be the activations per second of task i. At first we have to calculate the remote buffer accesses per second *raps*.

$$raps = \sum_{i=1}^{n} ba_i * P_i \tag{4.14}$$

Further we can calculate the remote accesses per second *rps* if the remote access overhead per access *raopa* is known :

$$rps = raps * raopa \tag{4.15}$$

Finally, similar to the calculations in Section 4.2.8.5, we can determine the overhead for remote buffer accesses normed to the core utilization as follows:

$$nmo = \frac{rps}{f} \tag{4.16}$$

The next sections shows the results of an analysis of a real example project, that is typical for automotive powertrain projects in terms of tasks, runnables and data accesses. Further, a comparison is made between buffer migration and remote access.

c) Results In this section we present the results of the analysis of a real automotive powertrain project. We examine the runtime overheads of migrating buffers on the one hand and remote access to buffered items on the other hand. We determine the overheads in terms of (core) utilization. At first we calculate the runtime overhead for migrating the buffers as it is described in Section 4.2.8.5.

The amount of buffered items *bi* (within the different schedule sections) can be extracted out of the project data, as well as the periods of tasks, P_i. As *opm*, an estimation for the runtime overhead per buffer migration, a value of 18.50 clock cycles * x (number of migrated elements) + 71.55 clock cycles is assumed. Where the additional term of 71.55 clock cycles is the average time that is necessary to initialize and terminate the buffer migration. This results from former measurements on an existing multiprocessor platform (Infineon AURIX TC27x) and is the average value for migrating a 16 Bit element.

Except for only a few buffers, most of them have the size of 16 Bit or lower, so this is a good starting point for our approximate examination (further, there are only marginal differences when migrating 8 Bit, 16 Bit or 32 Bit buffers). The clock frequency of the microprocessor is assumed to be 300 MHz.

Next, the runtime overhead for remote accesses to buffers is considered as described in Section 4.2.8.5. Generally, we assume a remote access overhead per access *raopa* from factor 2 for reading data and a factor of 3 for writing data in average (compared to a memory access to the core local RAM at the own core). Nevertheless, there might occur some events that slow down these processes, e. g. high bus traffic on the crossbar. So we vary the overhead factor in our examinations from 2 - 10. But as worst case, we assume that the overhead factor will never reach a level of > 6. Figure 4.19 shows the result of the examination.

Since we assume that the overhead factor for remote accesses will be less than 6, the application of remote accesses lead to a smaller runtime overhead than the migration-based approach. As a result, we decide to perform remote buffer accesses instead of buffer migrations in our concept. Nevertheless, migrating buffers stays an alternative option for future approaches. Additionally, when buffers are allocated on any of the core local RAM, the overhead factor becomes simply 1 if a job is executed exactly on this core. This was not considered in our determinations. But it is not only the runtime overhead which takes us to this conclusion. The buffer migration would also lead to

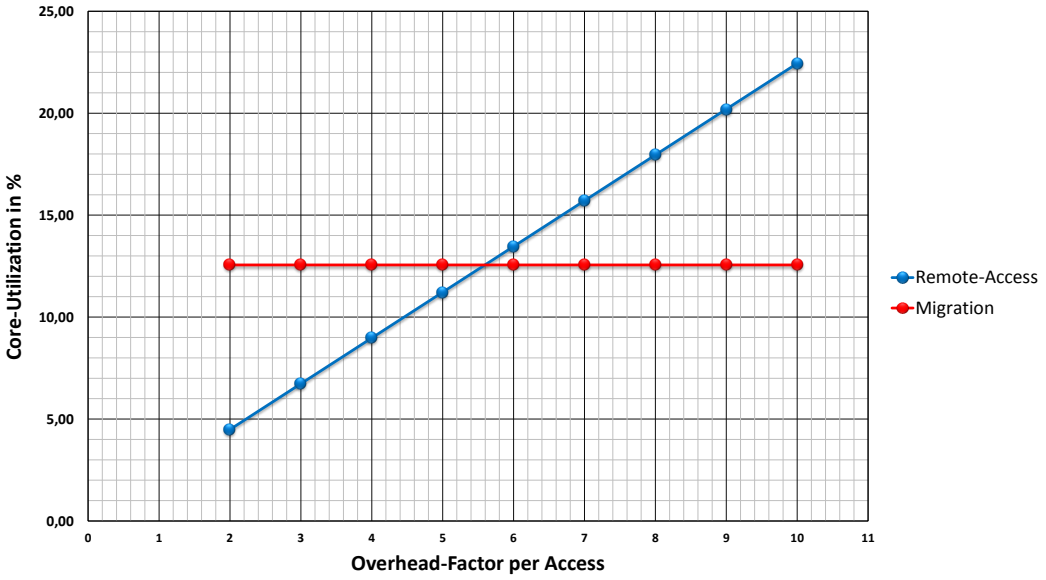

FIGURE 4.19: Remote Access vs. Buffer Migration. The core utilization (y-axis) is depicted over the overhead factor for remote accesses (x-axis).

a memory consumption which is m (core) times the memory consumption when using remote accesses, because for migrated buffers, memory has to be reserved on every core local memory.

4.2.9 System State Transitions

Besides the common task set there exist further special tasks in the area of automotive engine control systems, called system transitions. They are switching the major system wide operation modes which are managed by a state machine. When certain events occur during runtime, the engine state or engine control unit state may have to be changed, as for example if the engine is turned on by the vehicle driver. Transitions are synchronous events which are performed on all available cores simultaneously. Followed by this, the execution of common tasks has to be ramped down while a system transition is performed in order to guarantee that the transition is processed properly and the whole system stays consistent in the same, well defined state all the time. The existing transition concept is protected in the patent [1]. Up to now, state transitions within the partitioned task-fix priority scheduling are handled by a so called *event coordinator* which assures the ramp-down of the common tasks as well as the core synchronous execution of the system transitions. In the future, the handling of state transitions should be controlled by the EDF plug-in as well. This fact leads to some overhead for

the scheduler. It shall occur only if a transition event gets activated. When common tasks are proceeded without an activated transition event, there should not occur any additional overhead for the EDF plug-in. In the following sections, the structure of state transitions and its current proceeding approach (event coordinator) are described. Further, a new concept to integrate the handling of state transitions within the EDF module is presented.

4.2.9.1 Structure of System State Transitions

State transitions can be divided into different steps (A-D). Before describing these steps, we first introduce different types of task groups. All tasks within the task set are assigned to several groups according to their relative deadline. Those task groups indicate whether a task is allowed to execute in a certain step of a transition and whether it is important for the process of system transitions. The task groups are defined in Table 4.1.

TABLE 4.1: Overview and description of the different task groups. Every task in the whole task set belongs to one of the groups, so that state transitions can be performed properly.

Group 1	Long deadline tasks (delayed until step D if not already executed in transition step B1)
Group 2	Medium deadline tasks (executed if activated in step B1, blocked until step C2 has been finished)
Group 3	Short deadline tasks (executed if activated in step B2, blocked until step C1 has been finished).
Group 4	Very short deadline tasks (not disturbed in any step of transition).

Furthermore, the different (sequential) steps of state transitions are described in Table 4.2. An example schedule of such a state transition in case of uniprocessor is depicted in Figure 4.20. The schedule starts with a common execution of the task system, where tasks are scheduled according to their priorities (step A). Between time 2 and time 3, a need for a state transition is recognized (activation event for transition task *SysTran*), which means that the system transition starts its procedure with step B1. Within step B1, tasks that belong to group 1 (G1) are delayed if they get activated now (like task T_8

TABLE 4.2: Overview and description of the different transition steps.

Step A	Before transition detection; Standard behavior of tasks.
Step B1	Task ramp-down; Need for transition detected; Wait until all long and medium deadline tasks (group 1 and group 2 tasks) are terminated; Already activated, but not started tasks put 'on hold'; Short deadline tasks (group 3 and group 4) are authorized to execute.
Step B2	Total Task Ramp-down; Short deadline tasks are ramped-down (no more start allowed). Same behavior as B1, but for short deadline tasks (group 3).
Step C1	Full exclusive transition; No other task allowed except for group 4 tasks; All initializing functions which interfere with group 3 tasks have to be called in this phase.
Step C2	Partial exclusive transition; group 4 and group 3 tasks are allowed to be executed; All other initializing functions have to be called.
Step D	System recover; Pending tasks are released. When all memorized tasks have been terminated, go back to Step A.

at time 6). Further, tasks that are already running are allowed to finish their execution. At time 7, the last remaining G1 task is terminated which means that the transition enters step B2. If any task (G1 - G4) is activated within step B2, it is not allowed to start execution until step D is reached. Step B2 is finished, when all G3 and G4 tasks are terminated (at time 9). The transition now enters step C1 which means that the transition task *SysTran* itself is executed. During step C1, only G4 tasks are allowed to preempt the transition. Further, if all runnables within step C1 are processed at time 11, step C2 begins. The only difference to step C1 is that now G3 tasks are allowed to preempt the transition as well. Finally, at time 13, the complete transition is finished and step D is entered, which means that the common task execution is resumed. Note that there is no functional difference between step A and step D.

The applied transition handling (either the EDF plug-in or the event-coordinator) has to assure the different steps of a transition as well as the permissions of execution of (group 1 - group 4) tasks during a transition. The next section presents a concept of how runnables within transitions can be distributed over the available cores.

4.2.9.2 Static Concept For Transition Runnable Distribution

In the current parallel state transition concept, the handling of core synchronous state transitions is managed by a so called *Event Coordinator*. This Event Coordinator assures that the transition steps from A to D are performed correctly. Within the different

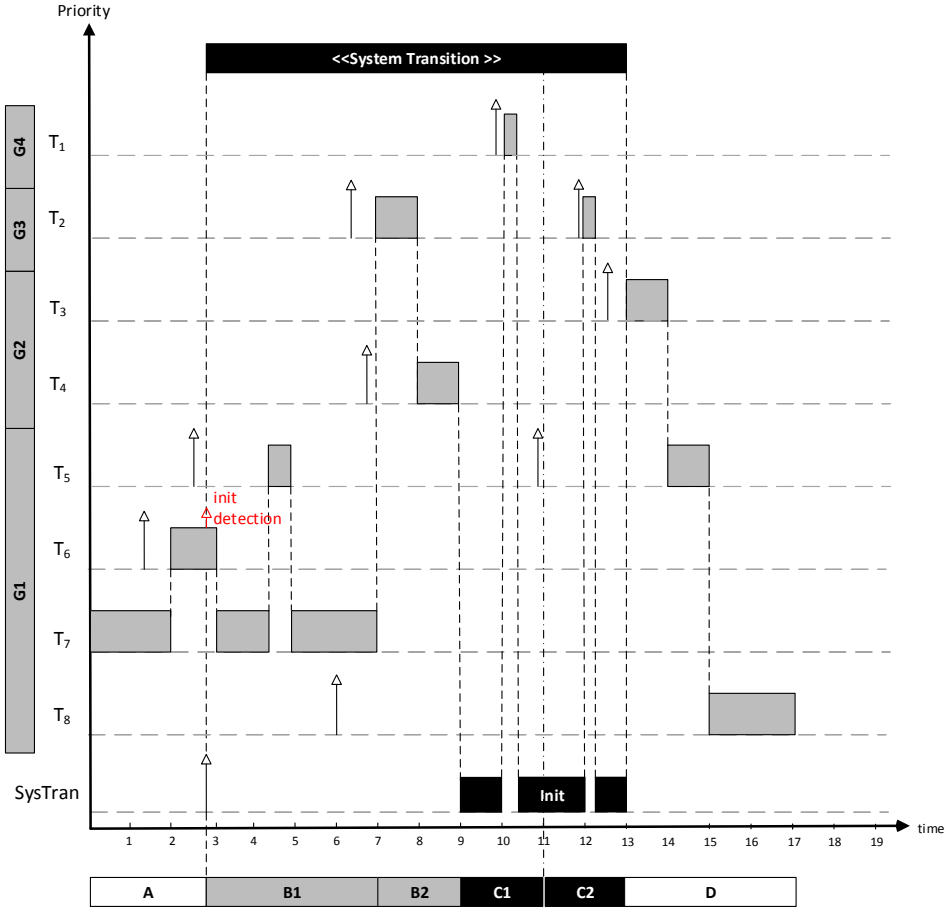

FIGURE 4.20: Example schedule of a state transition on a uniprocessor including the different steps from A to D. Task T_1 is a very short deadline task (group 4), task T_2 is a short deadline task (group 3), tasks T_3, T_4 and T_5 are medium deadline tasks (group 2) and T_6, T_7 and T_8 are long deadline tasks (group 1). Finally, SysTran is the transition task as itself.

steps of performing the state transitions in steps C1 and C2, different runnables have to be executed. These are initializing several data in order to set all necessary values for the new engine or engine control unit state. Since all cores are available during those transitions, it is possible to distribute the runnables of the transition tasks over the different cores to execute them in parallel. This approach has already been presented in [3]. In this transition concept, runnables that belong to system transitions are assigned statically to per core sequences and -steps at build time. On each execution, a transition always contains the same set of runnables in the same sequence. Therefore, an offline partitioning mechanism can be applied if the run times of the transition runnables are known.

In order to fulfill the constraints for data consistency in this special case, it is indispensable to consider the inner and outer dependencies of all transition runnables. We define inner dependencies by the data-flow between runnables within the transitions. If transition runnable S1 reads data that are produced by transition runnable S2, then S1 depends on S2. Moreover, outer dependencies arise through shared resources, that are accessed by transition runnable on the one hand, and further by tasks in group 3 or group 4 on the other hand. If resources are shared between group 1 or group 2 tasks and transitions, there is no danger for inconsistencies, because while a transition is performed (step C1 and step C2), group 1 and group 2 tasks are not allowed to execute. We next discuss how to assign transition runnables to their correct transition step and how to distribute those runnables onto the available cores.

a) Grouping Of Dependent Runnables As mentioned before, inner dependencies of transitions result from the data-flow between the runnables within the transitions. Figure 4.21 shows an example of different dependent runnables of a fictitious transition. In this example one can detect three layers of dependency: For instance, *Runnable 8* depends on *Runnable 5* and *Runnable 5* again depends on *Runnable 1*. To guarantee data consistency while proceeding a transition, such dependent runnables have to be executed in a strictly sequential order. Two different ways of building such a sequence are considered below.

i) Synchronization Points The first option to build correct sequences of dependent runnables can be realized by the use of synchronization points: Figure 4.21 shows how dependent runnables can be divided into different layers. Between these layers, *synchronization points* (also known as barriers) are placed.

After sorting the runnables in different layers the transition has to be performed in a way, that one layer after another is proceeded. As dependencies are divided through synchronization points, the sequence of runnables will always be correct. A drawback of this approach is, that runtime is wasted at every synchronization point, as all cores have to wait until the last one has executed its last runnable from the actual layer.

ii) Runnable Compositions As an alternative to synchronization points, so called *runnable compositions* can be built. An example for that is shown in Figure 4.22.

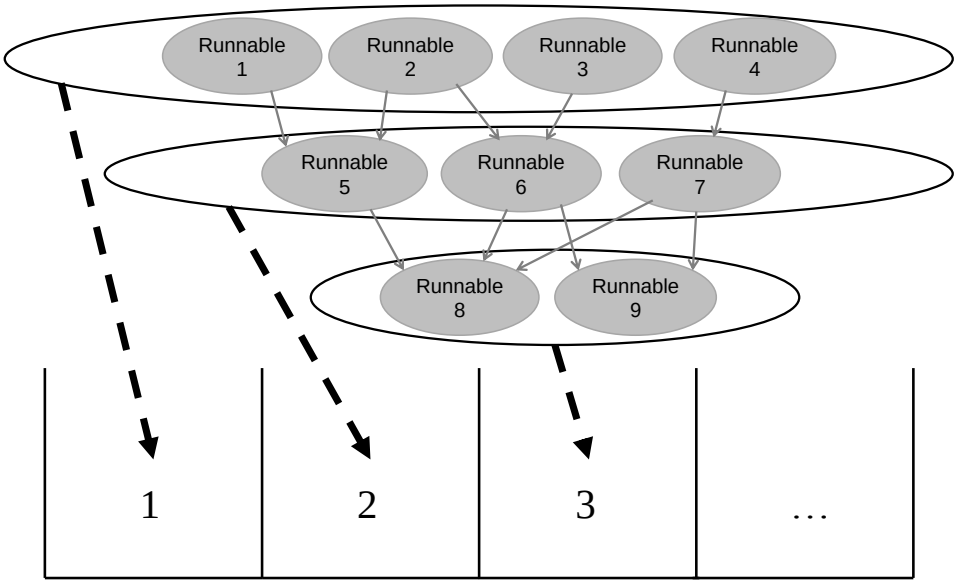

FIGURE 4.21: Ensuring requirements resulting from inner dependencies by different layers which are separated by synchronization points. The gray, solid-line arrows indicate dependencies between runnables caused by data-flow. The black, broken-line arrows symbolize the classification of the runnables to the layers. Synchronization points are set between the layers.

Dependent runnables (Runnable 1, Runnable 2 and Runnable 3 in this example) are combined to a *runnable composition* with a fixed sequence. Those compositions can now be seen as one top-level runnable that contains calls to all runnables within the compositions in a sequental order.

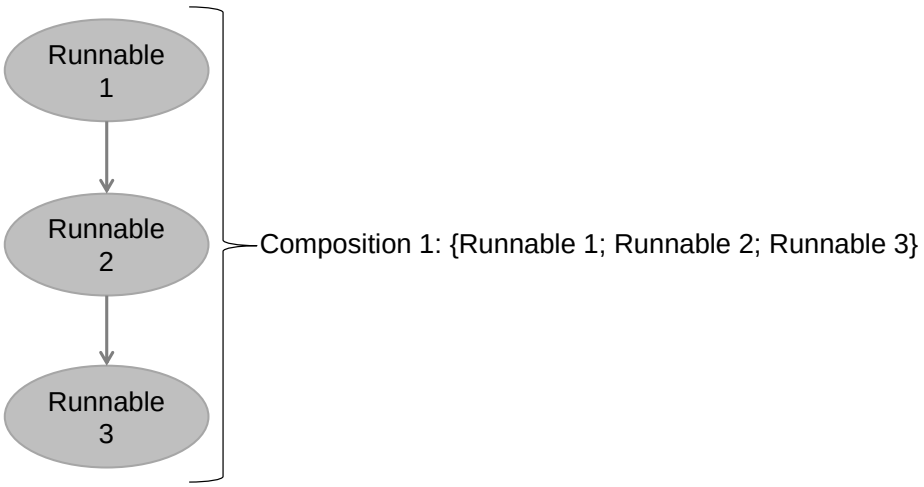

FIGURE 4.22: Ensuring requirements resulting from inner dependencies via runnable compositions. In this example, the runnables 1–3 which depend on each other are combined to composition 1.

The runtime of the runnable composition, which is important to know in order to distribute runnables to cores, is the sum of the individual contained runnables. The runnable compositions are independent from each other per definition and do not have any unresolved dependencies to another runnables within that transition any more.

Empirical investigations on real automotive powertrain projects showed: Transitions can be performed faster with runnable compositions than with synchronization points. This investigations have been already presented in [3]. Therefore, this approach is preferred to building synchronization points. Next, it is described how to assign transition runnables to the correct transition steps.

b) Assign Runnables To Transition Steps Because of the results in [3], we focus on runnable composition in order to solve inner dependencies of transition runnables. Further, if runnables are independent in terms of data flow, this single runnable builds its own composition.

The reason for dividing runnables into different steps lies in the fact that during step C1, tasks of group 3 are not allowed to execute (in contrast to step C2) and thus there can not occur any inconsistencies by simultaneously accessing the same data at one point in time.

If at least one of the runnables within a composition is accessing a data item which is accessed by any runnable in a task of group 3 or group 4 as well, the whole composition is assigned to step C1, otherwise to step C2.

When transition runnables or compositions respectively are assigned to the correct transition step, they can finally be assigned to a core for execution. This is described in the next sections.

c) Assign Transition Runnables To Cores Runnables of transitions shall now be assigned statically on a specific core for execution. At first, all compositions which were assigned to step C1 are distributed over the cores by applying the worst-fit decreasing bin-packing algorithm [8]. For this it is required to know the runtime of runnable compositions. The runtime can be evaluated either by measurement or by estimations (if no

[8] Different types of bin-packing were discussed to distribute runnables across the available cores. The results presented in [3] show that worst-fit decreasing leads to better results than other types of bin-packing.

measured runtime data are available).

Attention has to be paid to the fact that not all cores of the multiprocessor have necessarily the same processing speed, as this algorithm should be able to handle symmetric, uniform heterogeneous and unrelated heterogeneous multiprocessor architectures. Because of this, a performance index is introduced per core (this index depends on the processing frequency), so that the runtime on each core can be calculated correctly according to its processing speed. After that, the same approach is used again to distribute all compositions located in step C2.

4.2.9.3 Dynamic Concept For Handling System Transitions

With the development of the new scheduling concept the idea arises to make the event coordinator obsolete. Handle the system transitions can be taken over by the global scheduler. The main advantage of this approach lies in the possibility that transition runnables can be assigned dynamically to the cores. This allows a better load balance and thus shorter transitions compared to the static runnable assignment because the determined runtimes of transition runnables may vary from case to case. Further, no runtime measurement or estimation is required for transition runnables. However, the EDF plug-in has to make sure that transition steps are processed correctly, but produce no additional overhead when scheduling common tasks. The basic idea behind this is that usually, the common EDF scheduler plug-in is performed like described in the concept above. Further, only if a transition gets activated, the plug-in switches into the *transition mode*. The different transition steps have to be performed according to the definition in Section 4.2.9.1. How this can be realized correctly is discussed in the next section, followed by a concept for a dynamic distribution of transition runnables.

a) Realizing steps A - D of transitions The following table shows an overview of the different transition steps, in which steps which groups are allowed to execute or to be activated and what features are (additionally) necessary to manage the execution of transition steps correctly.

TABLE 4.3: Overview of different transition steps and what additional features are required within the EDF plug-in. A more detailed description of task groups can be found in Table Table 4.1, while transition steps are pointed out in Table Table 4.2.

Transition Step	A	B1	B2	C1	C2	D
Task Groups Allowed To Run	G1 - G4	G1 - G4	G2 - G4	G4	G3 + G4	G1 - G4
Allowed Activations	G1 - G4	G2 - G4	G4	G4	G3 + G4	G1 - G4
Necessary Features	• Recognition of Transition Event • Scheduler Mode Switch (Transition Mode)	$D_{sysTran} < D_{G1-Tasks}$	• Recognition G1 Tasks finished • G2 + G3 Tasks are not allowed to 'overtake' Transition	• Recognition G2 + G3 Tasks finished • Execute C1 Runnables	• G3 Tasks are allowed to 'overtake' transition again • Execute C2 Runnables	Switch back to common EDF mode

For transition step A, the plug-in has to be extended by two features. First, a recognition mechanism is required that detects transition activations. This could easily be added by inserting a flag to every task if it is a common task or a transition task. This would be the only overhead that affects the scheduler plug-in at every activation of any task. Further, if a transition task is detected, the plug-in has to switch in its transition mode.

Step B1 does only require that the relative deadlines of transition tasks have to be earlier than the deadlines of G1 tasks. EDF behavior assures the correct handling within step B1.

In contrast, transition step B2 requires a recognition that all G1 task are terminated now. Additionally, a kind of blocking mechanism has to be implemented that G2 and G3 tasks can not overtake transition tasks inside the global event list.

Transition step C1 requires a recognition (similar to that in step B2), that all G2 and G3 tasks are terminated (start criterion). Further, transition tasks have to be able to execute runnable compositions from C1 list within the transition event. This can be realized via chained lists and function pointers.

A mechanism that allows G3 tasks now to overtake (and thus preempt) the transition runnable is required. However, if the transition task has an earlier absolute deadline

than a G3 task, the transition task should continue execution. The execution mechanism of step C2 is the same as in step C1, but another task list is worked through.

Finally, in step D (if all C2 runnables are terminated), the scheduler plug-in has to switch back into the common execution mode. Thus, the transition has been finished.

b) Dynamic Concept For Transition Runnable Distribution As already mentioned, the processing of transition steps will be the same as in the static transition concept when the parts presented in Table 4.2.9.3 are realized. Like in the static concept with event coordinator, for each transition, a transition task is assigned on every core. The main difference from a runtime view of transition is that now the compositions are distributed dynamically over the cores.

Basically, a transition event is handled like a global scheduling event within the EDF scheduler plug-in. When a transition activation is detected, the scheduler switches into the transition mode. For each of the different steps of transition execution (C1 and C2), a global list exists that contains calls to all runnable compositions of the prevailing transition steps. It is still necessary to collect dependent runnables and put them into a composition because this way it can be assured that the sequencing constraints of runnables are kept. The single compositions are picked out of this global list and executed by the running transition tasks (if one composition is executed, the next one is selected) until the list is empty. The correct behavior of the transition steps is guaranteed by the features described in Section 4.2.9.3. When the first step (C1) of the transition is reached, the transition event tasks on the cores are executing all the runnable compositions within the global list of C1.

An example for this procedure is shown in Figure 4.23. On every core, an instance of the transition task is running. The transition tasks have each picked a runnable composition out of the global list of C1 composition and execute them (Composition 1 on Core 1, Composition 2 on Core 2 and Composition 3 on Core 3). If one of the executing compositions terminates, the next free (not executed) composition is picked from the global list and thus executed on the regarding core.

If C1 runnables were all processed successfully, list C2 is worked through. The management of which core has to execute which runnable is done via function pointer. If list C2 is done as well, the transition terminates and step D is reached (standard execution

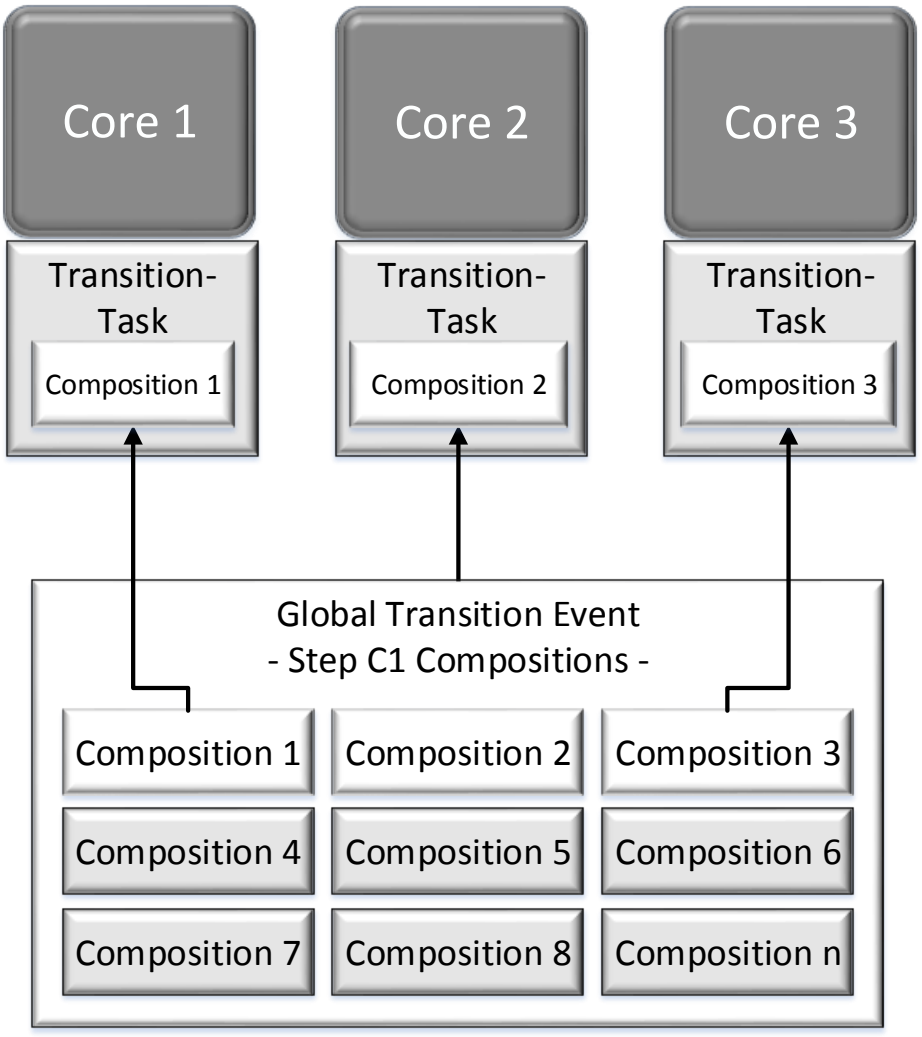

FIGURE 4.23: Performing a dynamically distributed transition (here step C1) where independent compositions are selected until the global composition list is empty.

of the system). The transition mode of the scheduler is terminated as well and the EDF plug-in switches back to standard EDF execution.

4.2.10 Parallel Tasks

Basically, schedule events (within the global event list) contain information about what schedule sections have to be executed by this activated job. In general, those schedule sections are executed in a sequential manner, as a job is always assigned onto one core (according to the EDF-schedule module). When this sequential model is applied, a schedule event represents a task.

Nonetheless, it might be useful to mark schedule sections that are independent from other schedule sections within that task in order to allow parallel execution. If a schedule event contains such units, these units can be sent to different cores for a parallel execution. One can take advantage of the concept of dynamically distributed transition described in Section 4.2.9.3. After all parts (schedule sections) of the global schedule event have been distributed, it is removed from the global list. The application of parallel (or partly-parallel) tasks may help to increase the load balance and thus the robustness of the system. Besides common sequential and parallel task, there exists a third type of task, called chained task. Chained tasks are discussed in the sections below.

4.2.11 Chained Tasks

Chained tasks are activated by the termination of their predecessor task: Namely if the reason for existence of such tasks lies in the physical constraints regarding two tasks. If a task requires to read data that are produced by another task, but they need strictly to be executed in succession. Two different approaches could be used to realize the concept of chained tasks within this scheduler concept. They are discussed in the following sections.

4.2.11.1 Additional Schedule Section as Chained Task

The first opportunity is to take one single schedule event (in global event list), but putting the chained tasks in two strictly sequential schedule sections. After the module *EDF-schedule* has been performed for the first of the chained tasks, it is not going to be deleted from the global event list, the successor task is just a further schedule section. The relative deadline of the chained task has to be set to the deadline of the last task of the chain which is here the end of the successor schedule section. Followed by this, no recalculation of the absolute deadline is necessary.

Nevertheless, this approach might be accompanied by the following drawbacks: First, if chained tasks remain in the global event list, there exists the probability that later parts of the chained tasks get EDF-activated while earlier parts are still executing (on a different core) and thus the required sequence cannot be kept any more (two chained tasks are running simultaneously on two different cores). Additionally, there might occur the problem that the termination tasks which are not at the end of a chain can not be recognized by the EDF-terminate function.

4.2.11.2 New Task as Chained Task

When the previous task within the chain terminates, it triggers automatically the activation of the successor task. For chained tasks, the schedule function is then started as it is the case for unchained tasks. Nevertheless it has to be guaranteed somehow that the first part of a chained task is not executed while a later part of a previous task instance is still running on a core. As a conclusion one can say, that successor tasks of chained tasks should be handled like a new task, just activated by a different kind of event trigger.

4.3 Efficient Multiprocessor Real-Time Synchronization Protocols

As mentioned in section 3.3, a high number of multiprocessor real-time locking protocols have been presented in the recent years. Nevertheless, all the described existing protocols are accompanied by some drawbacks. Either they do not permit nesting, or they can only be applied for a specific type of scheduling algorithm. The only protocol that allows nesting and supports all kinds of scheduling is the flexible multiprocessor locking protocol FMLP presented in [26]. However, due to the fact that non-preemptive busy-wait is applied within the short variant of the FMLP, a high priority task suffers from blocking even if it has no resource conflicts regarding the blocking (low priority) task. This means that there occur unnecessary blocking times and priority inversions.

In order to overcome this, we present a protocol called preemptable waiting locking protocol PWLP where the busy-wait phase of jobs is performed preemptively and thus the above described drawbacks can be reduced. Nonetheless, the PWLP still requires non-preemptive critical sections to avoid deadlock scenarios when nested resource requests are applied.

The second contribution in the field of locking protocols, the forced execution protocol FEP, is able to forgo non-preemptive critical sections. The idea behind that is that blocking should only occur if unavoidable. Therefore, it relinquishes non-preemptive critical sections and enables the busy-wait phase and critical sections to be preempted by tasks with higher priorities. The mechanism that provides progress to the system and thus avoids starvation is that the resource holding task gets priority boosted as soon as a further request for this resource is issued by another task. Followed by this, the resource holding task gets *forced* to execute and as a consequence the resource is released as quick as possible. Because of that, we called this approach forced execution protocol FEP.

Both, the PWLP and the FEP, are discussed in detail within the following sections. Later, in Chapter 5, case studies are presented in order to benchmark the PWLP and the FEP against the performance of the busy-wait variant of the FMLP.

4.3.1 The Preemptable Waiting Locking Protocol PWLP

The first contribution to synchronization is a busy-wait based protocol that uses preemptable waiting. Because of this property it is called preemptable waiting locking protocol PWLP. This work has already been presented in [2]. To the best of our knowledge, up to now, there exists no other preemptable waiting protocol that supports partitioned and clustered allocation as well as task-fix, job-fix and fully dynamic priorities and further nesting of resource requests. The PWLP combines all of these requirements. Basically, the PWLP can be seen as an extension of the short variant of the FMLP [26] that has already been discussed in Section 3.3.4.4.

After some necessary definitions and assumptions, a short reminder of the basic non-preemptive waiting technique of the FMLP-short is given. Followed by this, the PWLP is described in detail, including request rules, a description of deadlock freedom, a blocking analysis and finally an example schedule.

4.3.1.1 Assumptions and Definitions

We assume a SMP system consisting of m homogeneous processors with global shared memory. Compared to the basic task state model in Figure 4.2 in Section 4.2.1, a different task state model is required to realize the PWLP. Figure 4.24 shows the basic state model with the extension of a state called *polling*. This state is only added for a better understanding of busy-wait. A job $J_{i,j}$ that is in state *running* but has to busy-wait for a resource (another job $J_{a,b}$ owns this resource at this moment) is now said to be in state *polling*. As soon as the resource is released by job $J_{a,b}$, the request of job $J_{i,j}$ is satisfied and thus it continues executing in state running. Further, from now on, passive tasks states (e. g. *suspended*, *ready*) are shaded in grey, while active task states (e. g. *running*, *polling*) stay white. Within the next sections, the task state model of the PWLP is presented as well as a definition of shared resources within the PWLP.

a) Task State Model The task state model of the PWLP needs an additional state called parking. Thus, a job can be in one of the following states: *suspended*, *running*, *polling*, *parking* or *ready* as shown in Figure 4.25. Initially, all jobs are *suspended*. When an activation of event occurs for a job $J_{i,j}$, it gets activated and enters the state *ready*,

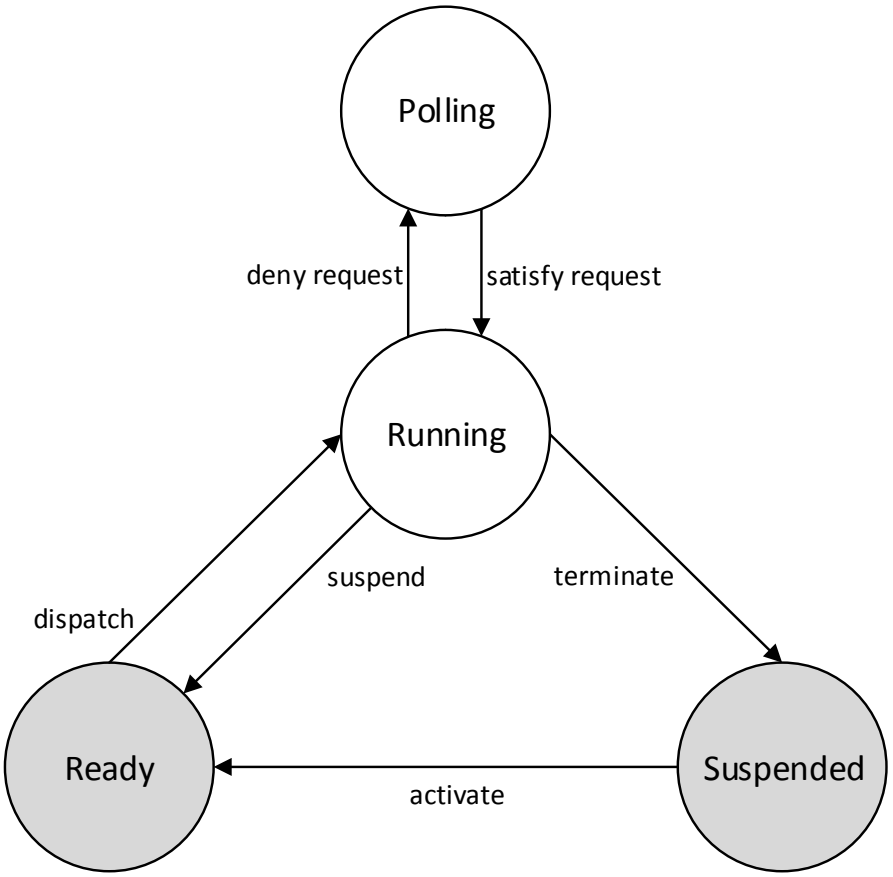

FIGURE 4.24: State diagram of a job $J_{i,j}$ containing state *polling*. White circles depict active states (a job is scheduled on a core), while grey circles mean passive states (tasks are *suspended*).

which means it is ready for execution. Further, if a ready job gets scheduled by the scheduler and starts executing, it turns into the state *running*. A running job $J_{i,j}$ may get *suspended* by the interruption of a job $J_{a,b}$ which has a higher priority. Then, $J_{i,j}$ returns to the state *ready*. After $J_{a,b}$ has finished its execution, $J_{i,j}$ is again *scheduled* and returns to the state *running*. While $J_{i,j}$ is *running*, it may require access to shared resources. Thus, $J_{i,j}$ will send a request to acquire the lock of the resource. If the resource request can be satisfied immediately, $J_{i,j}$ stays in the state *running* while it executes the critical section. Otherwise, if the resource is held by another job $J_{a,b}$, $J_{i,j}$ turns into the state *polling*, which means that $J_{i,j}$ remains scheduled on the core and busy-waits until the requested resource gets released. However, if the request is satisfied while $J_{i,j}$ is *polling*, the execution of the process is progressing again (it becomes *running*). Otherwise, if $J_{i,j}$ is *polling* and the job $J_{a,b}$ with higher priorities preempts $J_{i,j}$ while it

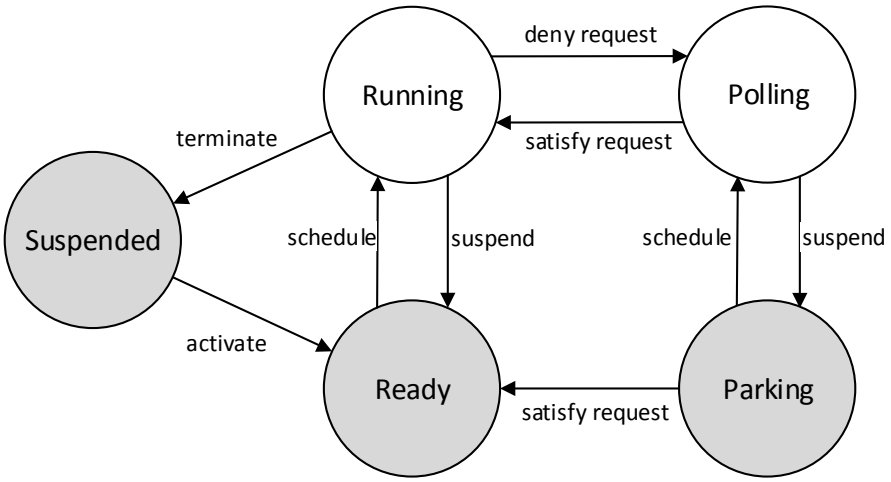

FIGURE 4.25: PWLP state diagram of a job $J_{i,j}$. Parking is added as an additional state of passive waiting.

busy-waits, $J_{i,j}$ gets suspended into the state *parking*. Later, $J_{i,j}$ becomes automatically *ready* if the required resource is released, so that $J_{i,j}$ may be scheduled again and thus returns to the state *running*. Next, if $J_{i,j}$ is *parking* and gets scheduled again, it resumes into the state *polling*. Finally, when $J_{i,j}$ finishes its execution, it gets *terminated* and remains suspended until the next job $J_{i,k}$ of task T_i is activated.

b) Shared Resources The PWLP allows nested requests to shared resources, as described in Section 3.3.4.1. However, deadlock avoidance has to be assured by applying rules regarding the order of requests, for example grouping of resource requests, like the FMLP uses it (see Section 3.3.4.4). Nevertheless, the PWLP also fits to different, less restrictive resource models, such as those described in [137]. In this resource model, a strict partial order of resource requests is required. This order is obligatory for every task, otherwise deadlocks may occur. For reasons of simplicity, we only consider mutex locks within the next sections. Note that in general it is also possible to apply other resource models like reader/writer distinction (see Section 3.3.1.1) as well.

4.3.1.2 Blocking Behavior of FMLP-Short

The PWLP is an extension of the short variant of the FMLP. Both protocols use busy-wait and FIFO ordered waiting queues per resource. Furthermore, both allow nesting if

a fitting resource model is obeyed. The main difference between FMLP-short and PWLP lies in the blocking behavior. The FMLP-short uses non-preemptable waiting when jobs poll for resources. In Figure 4.26, one can see that the non-preemptable section of the FMLP-short starts as soon as a resource request is issued. The critical section itself is executed non-preemptively as well. However, the fact that both, polling for a resource

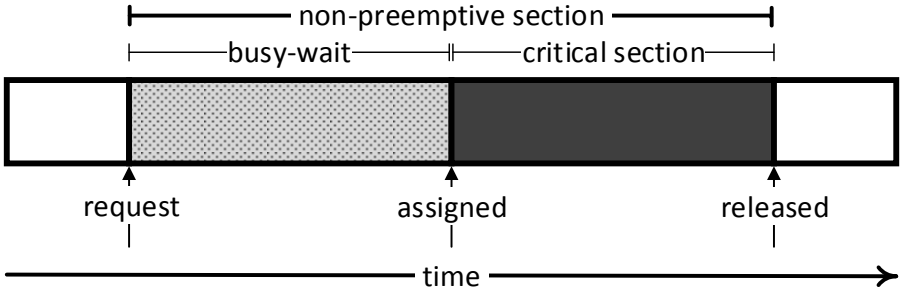

FIGURE 4.26: Basic structure of the different phases of the short variant of FMLP. Busy-wait and the critical section are non-preemptable.

and the critical section, are non-preemptable, may lead to more priority inversions than necessary. High priority tasks may be blocked while low priority tasks are polling for resources. As a result, deadlines of high priority tasks may be violated by unnecessarily long blocking times. A case study that confirms these assumptions is presented in [2].

4.3.1.3 Preemptable Waiting

In contrast to the busy-wait phases at the FMLP-short, our protocol takes advantage of preemptable FIFO-ordered locks. It is important to mention that this does not mean that critical sections are preemptable, too. Figure 4.27 depicts the corresponding blocking scenario from 4.26, but for the case when the PWLP is applied. After a resource request has been issued, the task is still preemptable, but as soon as the critical section is reached, the task becomes non-preemptable as well. Note that if some kind of nested resources are applied, the critical section starts at the point in time where the first resource request is satisfied. While waiting for resources, requesting jobs are preemptable by jobs who have higher priorities than the running job. Consequently, possible priority inversions are limited to the phases of critical section. In the following, the request rules of the PWLP are pointed out.

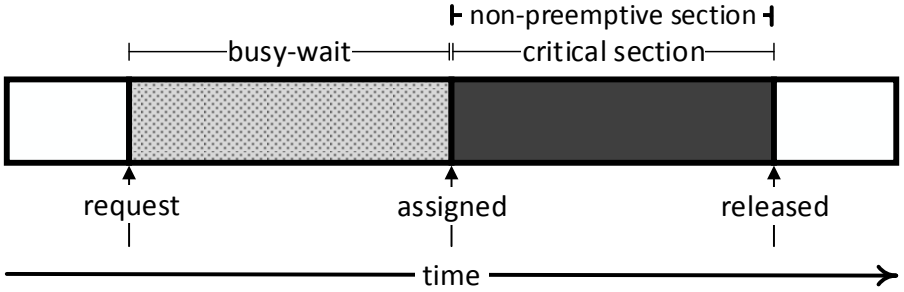

FIGURE 4.27: Overview of the different phases of the PWLP. Critical sections are executed non-preemptively, whereas busy-wait is executed preemptively.

a) PWLP Resource Request Rules The resource request rules specify how resource requests and releases are handled and what kinds of state transitions occur for jobs due to the locking behavior of the protocol.

Rule 1: If a job $J_{i,j}$ issues a request $Q_{i,l}$ for a resource R_l, it is added to the corresponding FIFO queue.

Rule 2: If the requesting job $J_{i,j}$ is in state running or polling, and the request $Q_{i,l}$ is the head of the FIFO queue, it is satisfied. Otherwise the access is denied.

Rule 3: If a resource R_l is released, the corresponding request $Q_{i,l}$ gets removed from R_ls FIFO queue.

Rule 4: If a resource request $Q_{i,l}$, issued by job $J_{i,j}$, is denied because it is not the head of the FIFO queue, $J_{i,j}$ becomes polling and thus busy-waits.

Rule 5: If job $J_{i,j}$ is in state polling, and another job $J_{a,b}$ gets activated with a higher priority than $J_{i,j}$, $J_{a,b}$ becomes running while $J_{i,j}$ gets suspended and enters the state parking.

Rule 6: If job $J_{i,j}$ is polling, and another job $J_{a,b}$ gets activated with a lower priority than $J_{i,j}$, $J_{i,j}$ continues busy-waiting in state polling while $J_{a,b}$ stays ready.

Rule 7: If resource R_l gets released by its owning job $J_{a,b}$, and $J_{a,b}$ continues running because it has a higher priority than $J_{i,j}$ ($J_{i,j}$ still polls for R_1 and is the head of the FIFO queue), $J_{i,j}$ changes its state from parking to ready.

Rule 8: If job $J_{i,j}$ changes its state from parking to ready according to Rule 7, $J_{i,j}$ is not allowed to acquire R_l. Later, when $J_{i,j}$ gets resumed and becomes running again, it is allowed to lock R_l.

b) Deadlock Freedom Remember the deadlock conditions of Section 3.3.2.2. Only one of the described conditions has to be broken in order to avoid such a scenario. Mutual exclusion and non-preemptable critical sections are present, that means that either the condition of hold and wait or rather circular wait has to be broken. Thus, the deadlock freedom of the PWLP can be proved by the following terms.

In case of non-nested resource requests, the hold and wait criterion is broken because a process can not hold a resource and wait for another resource at the same time per definition.

Furthermore, even if nesting is allowed, the circular wait criterion is broken by the PWLP. Deadlocks can occur if resource requests are satisfied by jobs which are in a passive state, either *ready* or *parking*. [9]

The PWLP avoids this case since resource requests are not satisfied when they change from *parking* to *ready* (see request Rule 8).

Jobs in state *ready* can not issue resource requests (by definition). Further, jobs in the state *parking* are not head of their FIFO queue (otherwise they either would not have to wait, or leave the state *parking* into the state *ready*).

Finally, if nested resource requests are applied, deadlocks may occur if those requests for resources are not grouped properly (see [26]), or if other resource model constraints are violated (e.g. those ones in [137]).

c) Blocking Analysis Similar to the short variant of the FMLP, two different types of blocking have to be taken into account. First, Busy-wait blocking is the interval of time a job busy-waits actively on a core. The second one is non-preemptive blocking, which denotes the time when a job $J_{a,b}$ is blocked due to the fact that another lower priority job $J_{i,j}$ performs a critical section. However, if a job $J_{i,j}$ performs passive waiting for a resource R_k in state *parking*, this is not a blocking time in the sense of non-preemptive blocking because even if R_k was available, $J_{i,j}$ would be not scheduled

[9] Tasks that are in state suspended are not active and thus do not have to be considered here.

at this moment. (Since $J_{i,j}$ is preempted at this point in time, but another job $J_{a,b}$ is *running* and followed by this, no processor time is wasted in contrast to busy-wait.) Nevertheless, passive waiting has an influence on the busy-wait blocking time, which is represented by the factor A_k in the following.

Busy-Wait Blocking. We denote $Tspin_{i,j}$ to be the maximum time a job $J_{i,j}$ is busy-waiting when it is not preempted by another job $J_{a,b}$. Analogue to the blocking analysis in [138] for preemptable FIFO spin-locks, busy-wait blocking can be described as follows: Let A_k be the maximum number of times that a request for resource R_k (or a nested group of resources) might be preempted by another job with higher priorities. Furthermore, a trivial bound on the sum of A_k is given by the number of jobs with higher priorities which are possibly released while $J_{i,j}$ busy-waits. Then one can sum up the maximum busy-wait blocking $mBWB$ of a task T_i as follows:

$$\mathrm{mBWB}(T_i) \leq A \cdot \sum_{\mathcal{R} \in Q} \mathrm{Tspin}_{i,j} \tag{4.17}$$

where Q is the whole set of resource requests that are issued by job $J_{i,j}$.

Non-preemptive Blocking. A job $J_{i,j}$ suffers from non-preemptive blocking if it is executing actively on a core but gets blocked because another job with a lower priority $J_{a,b}$ has acquired a resource Rk and thus performs its critical section $cs_{a,k}$. The maximum non-preemptive blocking for a task T_i occurs while the longest critical section of all tasks with a lower priority than T_i is executed. This is described by equation 4.18. B denotes the set of tasks which contains all tasks with a lower priority than task T_i, and δ(B) is the corresponding non-preemptive execution time (i.e. length of the critical section at the PWLP).

$$\mathrm{NP}(T_i) \leq max\{\delta(B)\} \tag{4.18}$$

In contrast to the FMLP, the time of spinning has no influence on non-preemptive blocking so that the non-preemptive blocking is reduced by the PWLP. This is caused by the fact that busy-wait is preemptable within the PWLP.

4.3.1.4 PWLP Example Schedule

To give the reader a better insight to the different behaviors of the FMLP and the PWLP, we present in the following a scheduling example, depicted in Figure 4.28. We

assume two processors where partitioned scheduling is applied for the execution of three different tasks. Task T_1 and task T_2 are assigned to processor P_1 while Task T_3 is assigned to processor P_2. T_1 and T_3 both access the shared resource R_1 while T_2 does not access any resources. T_1 has the highest priority, followed by T_2 and T_3 with the lowest priority. T_3 is activated at time 1, T_2 at time 2 and finally T_1 at time 3.

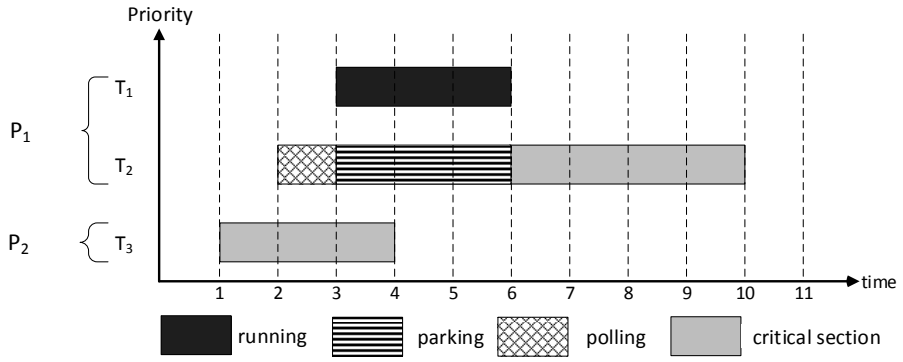

FIGURE 4.28: PWLP example schedule for partitioned scheduling. Task T_2 gets preempted by the higher priority task T_1 while it waits for a resource held by task T_3.

Task T_2 starts polling when it is activated on processor P_1 at time 2. At time 3, Task T_1 is activated on processor P1 and preempts Task T_2 because Task T_1 has a higher priority. Thus, Task T_2 becomes *parking* which means that it passively waits. When Task T1 is terminated at time 6, Task T_2 becomes resumed and gets *running* again because meanwhile Task T_3 has released resource R_1 (If Task T_3 had not released R_1 at time 6, Task T_2 would change to *polling* again).

4.3.2 The Forced Execution Protocol FEP

As a second locking protocol contribution, we present the Forced Execution Protocol FEP. The main objective of this protocol is to block only if necessary. Initially, busy-wait and the critical section itself are preemptable. Nevertheless, a task T_i that owns a resource R_l gets priority-boosted if a further task T_a issues a request $Q_{a,l}$ for resource R_l meanwhile. Followed by this priority boost, task T_i gets forced to execute in order to release R_l as quickly as possible, so that task T_a is able to make progress in the following. The main idea behind that is, that no task gets blocked unnecessarily and thus the schedule is disturbed as little as possible. The following sections provide assumptions and definitions for the FEP, as well as a description of the blocking behavior and an example schedule.

4.3.2.1 Assumptions and Definitions

In general, a similar task state model as it has already been defined for the PWLP in Section 4.3.1 can be applied for the FEP. However, the only difference is that under the FEP, a *parking* task is not allowed to be scheduled. That means that state transitions between *parking* and *polling* are not allowed anymore. The reason for that is that scheduling of tasks in state *parking* leaves a task poll for a resource and as a consequence, this task is not able to make progress. Instead of this, another task might be executed without a denied resource request. The model of shared resource can be kept completely equally to that one of the PWlP in section 4.3.1.

4.3.2.2 Blocking under FEP

Under FEP, not only preemptable waiting is applied, but even critical sections are preemptable. Resource requests are inserted into a FIFO ordered per resource wait-queue, where access to the prevailing resource is granted to the head of the queue.

Further, if more than one resource request is listed in a queue, the job that has issued the head request (the resource owning job) gets priority boosted. Remember that priority boosted jobs have higher priorities than all the other non-boosted jobs. Besides, priority boosted jobs are sorted according to the applied scheduling rules. Unbounded blocking times can be avoided by taking advantage of priority-boosting.

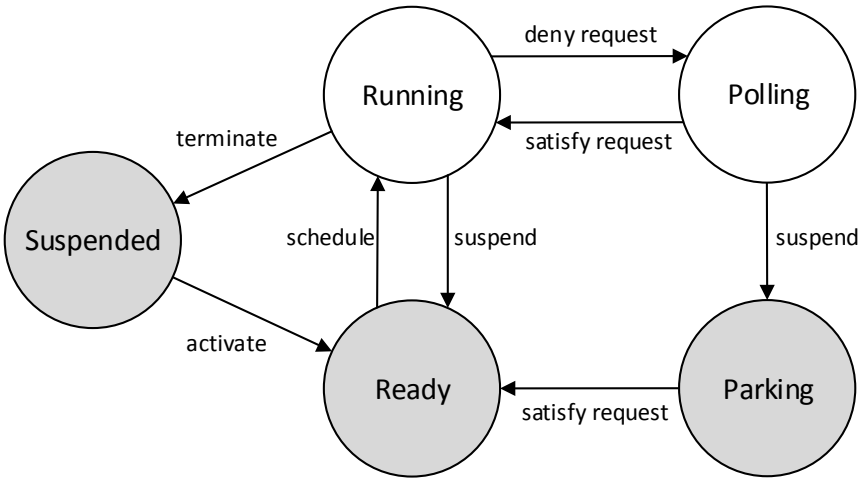

FIGURE 4.29: FEP state diagram of a job $J_{i,j}$. Transitions from *parking* to *polling* are not possible in comparison to the PWLP.

a) FEP Resource Request Rules The resource request rules specify how resource requests and -releases are handled under FEP and what kinds of state transitions occur for jobs due to the locking behavior of the protocol.

Rule 1: If a job $J_{i,j}$ issues a request $Q_{i,j}$ for a resource R_1, it is added to the corresponding FIFO queue.

Rule 2: If the requesting job $J_{i,j}$ is in state *running* or *polling*, and the request $Q_{i,l}$ is the head of the FIFO queue, it is satisfied. Otherwise the access is denied (If $J_{i,j}$ is *parking*, $Q_{i,l}$ is skipped and the request $Q_{a,l}$ of the next job in the queue that is *running* or *polling* is satisfied).

Rule 3: If a resource request $Q_{i,l}$ issued by a *running* job $J_{i,j}$ is denied because it is not the head of the FIFO queue, $J_{i,j}$ becomes *polling* and thus busy-waits. Followed by this, a job $J_{a,b}$ that is head of the wait-queue becomes priority boosted. Furthermore, the scheduler which $J_{a,b}$ is assigned to is called and thus $J_{a,b}$ is forced to execute and finish its critical section.

Rule 4: If a resource R_l is released, the corresponding request $Q_{i,l}$ gets removed from R_ls FIFO queue. Further, if a job is priority boosted because of owning resource R_l that is released at this moment, job $J_{i,j}$ returns to its standard priority.

Rule 5: If a resource R_l is released according to Rule 4, and the wait-queue still contains more than one waiting jobs, the new head of the wait-queue does not only get access to the resource, but it additionally gets priority boosted.

Rule 6: If a job $J_{i,j}$ is in state *polling*, and another job $J_{a,b}$ gets activated with a higher priority than $J_{i,j}$, $J_{a,b}$ becomes *running* while $J_{i,j}$ gets suspended and enters the state *parking*.

Rule 7: If an executing job $J_{i,j}$ is in the state *polling* and another, priority boosted job $J_{a,b}$ is head of the ready list, $J_{i,j}$ gets *suspended* (no matter if $J_{i,j}$ is priority-boosted or not) and $J_{a,b}$ becomes dispatched and thus enters state *running*. As a consequence, the progress of the system is ensured.

Rule 8: If a priority boosted job $J_{i,j}$ is in state *parking*, it is not considered for scheduling (Otherwise this would be a transition from *parking* to *polling*, and $J_{i,j}$ would not make progress anyhow).

Rule 9: If job $J_{i,j}$ is in state *parking* (issued a request $Q_{i,l}$) and the resource R_l gets released by the resource holding job $J_{a,b}$, job $J_{i,j}$ becomes *ready*.

Rule 10: If job $J_{i,j}$ changes its state from *parking* to *ready* according to Rule 7, $J_{i,j}$ is not allowed to acquire R_l. Later, when $J_{i,j}$ gets resumed and becomes *running* again, it is allowed to acquire R_l.

b) Deadlock Freedom Circular wait can be avoided by two properties of the FEP. First, if necessary, resource holding jobs are priority boosted in order to make progress and finally release the accessed resource. Further, jobs in state *polling* are preempted by the scheduler, if any priority boosted job is ready for execution but not in state *parking*. Furthermore, in case when nesting of resource request is allowed, the same resource models as described in Section 4.3.1.3 can be applied. Followed by this, the FEP is deadlock free as well.

d) Blocking Analysis In contrast to the PWLP, no non-preemptive blocking occurs under the FEP since busy-wait and critical sections are both preemptive. The following analysis is presented under the assumption of non-nested resource requests. The

maximum waiting time for a resource k can be described as follows:

$$\mathrm{mWB}(\mathrm{T}_i) \leq \mathrm{A} \cdot (\delta_{A,k}+\mathrm{Tschedule}_{A,k}) + (\mathrm{B} \cdot \mathrm{Tboost}(\mathrm{T}_b)) \tag{4.19}$$

Where A denotes the set of resource requests that are issued before the request of task T_i and still remain in the FIFO wait-queue when task T_i issues a resource request. The duration of the longest critical section of a task in A with resource k locked is described by $\delta_{A,k}$. Further, the term *Tschedule* contains overheads for scheduling, dispatching and satisfying the resource request has to be considered for any of the A requests.

Further, a critical section of a task within A, $\mathrm{CS}_{A,k}$, can be preempted by a task within the set B. B contains all tasks that are priority boosted while task T_i is spinning and further it has a higher basic priority than the current resource holding task. Note that the current resource holding task is priority boosted as well, since at least task T_i is additionally waiting for resource k. Followed by this, the actual resource holding task can be only preempted by another priority boosted task T_b, if $\mathrm{Y}_b > \mathrm{Y}_i$. Of course, if task T_b's priority gets resumed back to its basic priority after $\mathrm{Tboost}(\mathrm{T}_B)$, the currently resource k holding task gets scheduled again, except a further task within set B has a higher basic priority. $\mathrm{Tboost}(\mathrm{T}_B)$ here stands for the maximum duration a task within B is priority boosted.

4.3.2.3 FEP Examples

This section describes an example schedule of FEP under partitioned scheduling and additionally some examples of how requests are handled within the wait-queues of resources.

a) Example Schedule The simple FEP example schedule in Figure 4.30 shows three tasks partitioned onto two processors. Task T_2 starts executing at time 1 on processor P_1 and enters a critical section (resource R_l) at time 2. Later, at time 4, the high priority task T_1 gets activated on processor P_1 and preempts T_2. At the same time, task T_3 gets activated and starts execution on processor P_2. T_3 issues a resource request $\mathrm{Q}_{3,l}$ to that resource that is held by T_2. Since $\mathrm{Q}_{3,l}$ is denied and $\mathrm{Q}_{2,l}$ is head of the wait-queue of resource R_l, task T_2 gets priority boosted. As a consequence, the priority boosted task T_2 preempts the running task T_1 on processor P_1. At time 8, task T_2 releases resource

R_l, that means that T_2's priority gets restored to the original value. Thus, high priority task T_1 again preempts T_2 and continues execution. Besides, as resource R_l becomes available, request $Q_{3,l}$ is satisfied which means that task T_3 is now able to perform the critical section.

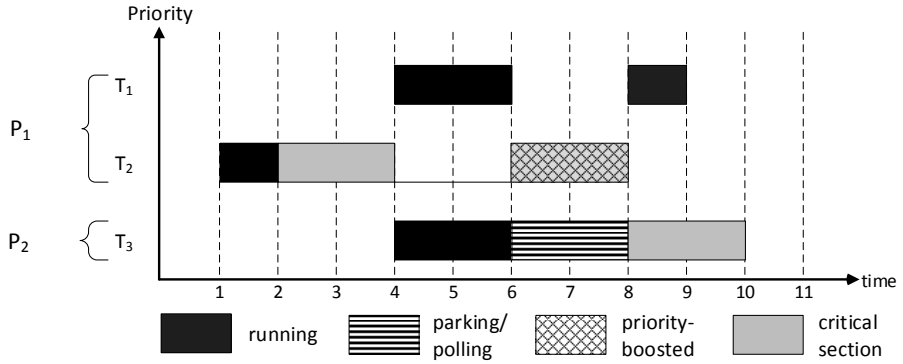

FIGURE 4.30: FEP example schedule. Task T_2 gets preempted during a critical section by higher priority task T_1. Later at time 6, task T_3 issues an additional request to the resource acquired by T_2, and thus T_2 gets priority boosted and forced to execute.

b) FEP Wait-Queue Examples In addition to an example schedule we present some use cases of the wait-queue at the FEP. The first example is shown in Figure 4.31 and depicts a wait-queue in two successive situations. First, in part (a), only the request $Q_{1,l}$ of the resource holding job is inside the wait-queue. Later, in part (b), if a further request $Q_{2,l}$ is issued, the resource holding task T_1 gets priority boosted. The intention behind this is that now task T_1 is able to release R_l as soon as possible so that request $Q_{2,l}$ can be satisfied quickly.

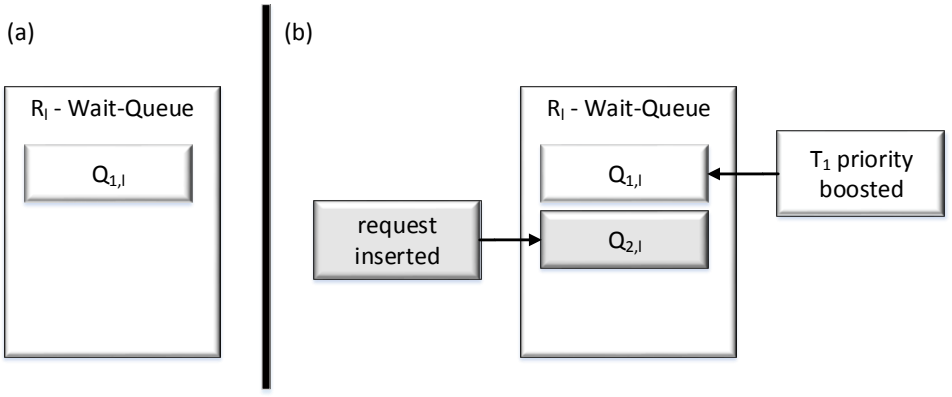

FIGURE 4.31: (**a**) R_ls wait-queue of resource contains only the request of the lock holding task T_1. (**b**) An additional request $Q_{2,l}$ from another task is inserted into the list, thus task T_1 gets priority boosted.

Figure 4.32 shows a scenario where more than one request is listed in the wait-queue and the resource holding process is releasing the corresponding resource. Task T_1 is in state *runnin*g and releases resource R_l. Followed by this, the priority of T_1 gets restored to its original value and the request $Q_{1,l}$ is removed from the wait-queue. After that, the next request in the wait-queue, $Q_{2,l}$, is satisfied and gets priority boosted because there is an additional pending request $Q_{3,l}$ listed in the queue.

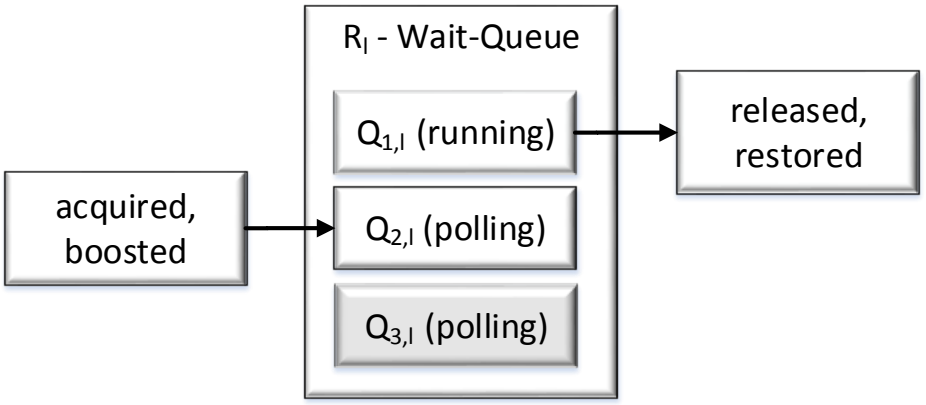

FIGURE 4.32: Wait-queue scenario at release with more than one pending request. The priority of the releasing task T_1 is restored to the original value, whereas task T_2 of the (next) satisfied request is priority boosted.

Finally, Figure 4.33 shows a similar use case like the example above (Figure 4.32). The difference lies in the state of task T_2 that has issued a resource request $Q_{2,l}$. Since T_2 does not have a sufficient priority, it is suspended and thus remains in state *parking*. According to Request Rule 2, this request $Q_{2,l}$ is skipped, and $Q_{3,l}$ is satisfied, since its task T_3 is in the active state *polling*. Note that if only tasks in state *parking* are listed in the queue, the resource is not granted to any of them. Instead of this, those tasks change their state from *parking* to *ready* as the resource is not held by any task now. Further, as soon as task T_2 gets resumed at a later point in time, its resource request $Q_{2,l}$ is satisfied.

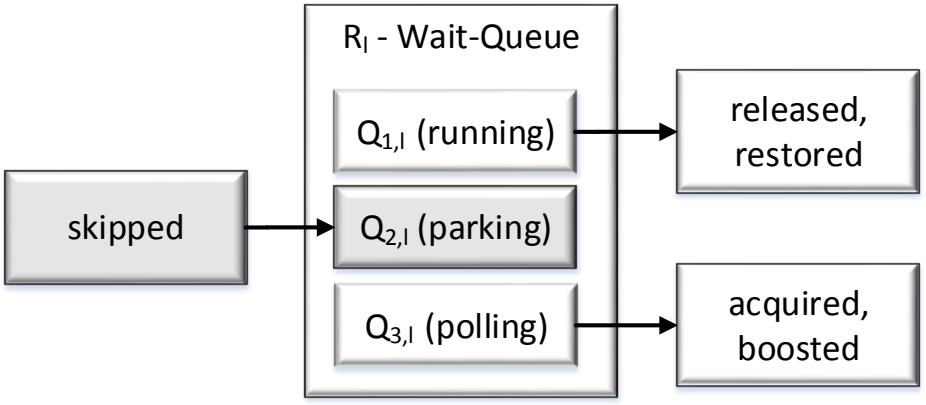

FIGURE 4.33: Scenario where request $Q_{1,l}$ is released and the wait-queue contains further requests. Followed by the release, the request $Q_{2,l}$ is skipped because the corresponding task T_2 is in state *parking*. Finally, request $Q_{3,l}$ is satisfied and priority boosted.

Chapter 5

Case Studies

This chapter provides a selection of case studies that are performed to evaluate the robustness of the locking protocols presented in the contribution of this thesis (see Chapter 4). The evaluation of protocols is carried out via event based simulation [135]. The first two experiments compare the protocols PWLP and FEP with the often discussed FMLP [26]. The FMLP is a suitable locking protocol because it has similar properties (it supports all types of scheduling and nesting of resource requests) like the PWLP and FEP. Further, it has been shown that the FMLP outperforms protocols like the M-PCP and D-PCP in [15] and also the M-SRP in [26].

The case studies are performed by task sets that are randomly generated with sporadic activated tasks and implicit deadlines. Every generated task set is simulated with each protocol and analyzed regarding the maximum of its *Normalized Response Time* NRTmax. This benchmark metric is used for analysis because it describes whether a task set kept all its deadlines during the whole simulation. The next section presents a more detailed description of NRTmax. Followed by this, two case studies are presented that use randomly generated task sets.

5.1 Evaluation Metrics

The main motivation to develop synchronization protocols was to rise robustness of embedded multiprocessor real-time systems. The metric that gives an evidence on keeping

all the deadlines is the Normalized Response Time NRT, that has already been introduced in Section 2.6.3.3. Since we want to evaluate whether a task set is schedulable under application of a certain synchronization protocol, we have to find out the maximum Normalized Response Time NRTmax.

Recall that the NRT of a job $J_{i,j}$ is denoted as the response time divided by the relative deadline of the job $d(J_{i,j})$:

$$NRT(J_{i,j}) = \frac{RT(J_{i,j})}{d(J_{i,j})} \tag{5.1}$$

If $NRT(J_{i,j}) < 1.0$, the deadline of job $J_{i,j}$ is kept, otherwise it is violated. The NRTmax of a task T_i is the maximum of all NRT values of all of T_is jobs:

$$NRTmax(T_i) = max_{j=1...g}(NRT(J_{i,j})) \tag{5.2}$$

Finally, one additional step is required to find the NRTmax of a whole task set τ. For that, the maximum of the NRTmax over all tasks within τ has to be found:

$$NRTmax(\tau) = max_{i=1...N}(NRTmax(T_i)) \tag{5.3}$$

$NRTmax(\tau)$ is the maximum NRT of the whole task set. If its value is smaller than 1.0, no deadline in the whole task set is violated and thus the task set is schedulable during the whole simulation duration. As a further metric, the ratio of schedulable task sets denotes the ratio of how many task sets have a $NRTmax(\tau) < 1.0$ and thus were schedulable within the executed experiment.

5.2 Synchronization In A Quad-Core System With Moderate Synchronization Overhead

The first experiment is a benchmark of locking protocols for a quad-core processor. We vary the number of resource accesses k and thus the number of critical sections per task. As a result, the synchronization overhead increases with the number of resource accesses per task. The term *moderate* means that despite the synchronization overhead, the system is able to keep all the deadlines of its task set an thus remains robust. The next sections present the experimental settings and results of this case study.

5.2.1 Experimental Settings

We assume a symmetric multiprocessor system containing four cores with a clock frequency of 2.7 GHz each. We assume that one instruction per clock cycle is performed by each processing core. Every randomly generated task set contains 30 tasks, each with sporadic activation and implicit deadlines. The sporadic activations occur in periods between [10 ms - 100 ms], the next task activation is calculated by uniform distribution during the simulation run (online).

The utilization of tasks lies in the range between [1.0 % - 10.0 %] according to uniform distribution. As we apply partitioned EDF scheduling within this case study, tasks are assigned to cores by worst fit decreasing bin-packing WFD (see description in Section 3.2.3.1). Based on the task utilization, the number of instructions of the task can be derived.

Runnables are generated with instruction numbers by Weibull distribution in a range between [min: 100.000; avg: 500.000; max: 1.000.000], until the task utilization is reached. After the creation of all tasks, a feasibility check is performed to sort out task sets that are not schedulable under any algorithm, without considering synchronization overheads. Task sets that are not feasible are thrown away and a new one is generated instead (those task sets should not be examined in these experiments).

Resource accesses per tasks are varied in the range [0 - 22]. The critical section length is defined by the runnable, which is chosen for each resource access randomly. The specific resource that is accessed by a resource request is selected randomly from the pool of generated resources (draw with returning, since nesting is allowed). The number of resources N_r, that have to be shared between the tasks is calculated according to Equation 5.4.

$$N_r = \frac{k \cdot N}{\alpha \cdot m} \tag{5.4}$$

Where k denotes the number of resource accesses per task, N denotes the number of tasks, α the sharing degree and m the number of processing cores. In this experiment, we choose $\alpha = 2$. We consider only mutual exclusion as resource model (description see Section 3.3.1.1) and a requesting overhead (access costs) in the range of [1.0 μs - 15.5 μs]. This assumption corresponds with the results in [30]. For each number of resource accesses per task, 1000 simulation models are generated. For this case study, this means that 69 000 simulation runs are performed with a simulation duration of 20 s (23 000

task sets simulated with 3 synchronization protocols each). The simulation duration is determined empirically in order to get a trade-off between reaching the maximum values of NRTmax and keeping the simulation time in a feasible time.

5.2.2 Results

Figure 5.1 depicts the results of the case study. The first part shows the ratio of schedulable task sets for FMLP, PWLP and FEP. One can see that in this experiment, all task sets were schedulable, that means that no task has violated its deadline during the whole simulation duration over all levels of k.

The other pictures in Figure 5.1 show boxplots of the NRTmax(τ) of all simulated task sets, performed with either FMLP, PWLP or FEP. The NRTmax of FMLP and PWLP have almost equal values, this can also be seen in Table 5.1. In contrast, the FEP performs better regarding NRTmax, its value remains almost constant and lower than the NRTmax of FMLP and PWLP over all values of k. The tables 5.1 and 5.2 show the results more in detail. For all task sets of every k, the minimum, average and maximum values of NRTmax are presented.

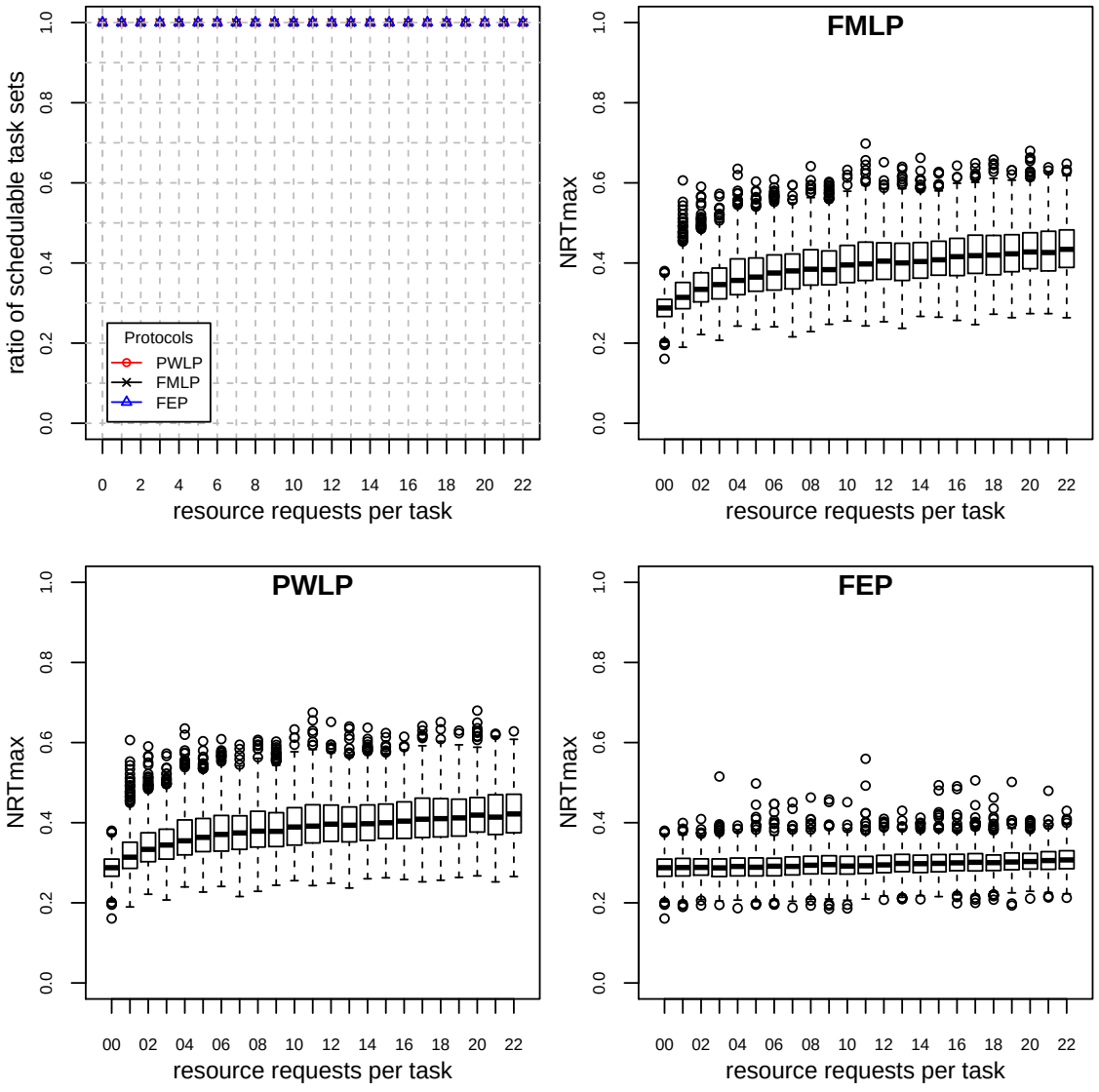

FIGURE 5.1: Results of case study 1. The upper left picture presents the ratio of schedulable task sets for all protocols. The upper right and both pictures at the bottom show the NRTmax for task sets simulated with either FMLP, PWLP and FEP respectively. The x-Axes show the resource requests per task, while the y-Axes present the ratio of schedulable task sets or rather the values of NRTmax.

TABLE 5.1: Maximum values of normalized response time of FMLP, PWLP and FEP for k = 0 - 11 in case study 1.

k	00	01	02	03	04	05	06	07	08	09	10	11
max_FMLP	0.3800	0.6062	0.5906	0.5725	0.6352	0.6034	0.6086	0.5948	0.6417	0.6026	0.6326	0.6980
max_PWLP	0.3800	0.6062	0.5906	0.5725	0.6352	0.6034	0.6086	0.5948	0.6068	0.6026	0.6329	0.6281
max_FEP	0.3800	0.3995	0.4091	0.5152	0.3930	0.4979	0.4468	0.4509	0.4628	0.4579	0.4512	0.4297
avg_FMLP	0.2877	0.3243	0.3430	0.3523	0.3680	0.3750	0.3817	0.3827	0.3912	0.3913	0.3996	0.4370
avg_PWLP	0.2877	0.3237	0.3423	0.3504	0.3660	0.3726	0.3778	0.3786	0.3860	0.3863	0.3939	0.4253
avg_FEP	0.2877	0.2883	0.2893	0.2881	0.2899	0.2900	0.2915	0.2927	0.2946	0.2952	0.2940	0.3076
min_FMLP	0.1609	0.1897	0.2214	0.2071	0.2426	0.2346	0.2408	0.2157	0.2287	0.2470	0.2553	0.2635
min_PWLP	0.1609	0.1897	0.2214	0.2071	0.2396	0.2268	0.2408	0.2157	0.2287	0.2438	0.2553	0.2657
min_FEP	0.1609	0.1898	0.1936	0.1947	0.1866	0.1952	0.1955	0.1880	0.1931	0.1847	0.1859	0.2126

TABLE 5.2: Maximum values of normalized response time of FMLP, PWLP and FEP for k = 12 - 22 in case study 1.

k	12	13	14	15	16	17	18	19	20	21	22
max_FMLP	0.6516	0.6402	0.6623	0.6273	0.6431	0.6491	0.6583	0.6319	0.6798	0.6389	0.6480
max_PWLP	0.6751	0.6516	0.6402	0.6370	0.6239	0.6147	0.6413	0.6512	0.6299	0.6798	0.6220
max_FEP	0.5594	0.4097	0.4299	0.4126	0.4936	0.4909	0.5055	0.4625	0.5017	0.4060	0.4797
avg_FMLP	0.4058	0.4080	0.4073	0.4105	0.4159	0.4186	0.4227	0.4234	0.4263	0.4332	0.4319
avg_PWLP	0.3993	0.4012	0.4003	0.4029	0.4067	0.4091	0.4136	0.4137	0.4156	0.4225	0.4210
avg_FEP	0.2956	0.2965	0.2989	0.2978	0.2996	0.3010	0.3023	0.3008	0.3035	0.3045	0.3065
min_FMLP	0.2430	0.2534	0.2370	0.2667	0.2648	0.2567	0.2462	0.2722	0.2635	0.2736	0.2736
min_PWLP	0.2430	0.2491	0.2370	0.2600	0.2625	0.2581	0.2524	0.2563	0.2634	0.2673	0.2522
min_FEP	0.2097	0.2075	0.2090	0.2087	0.2155	0.1986	0.1997	0.2078	0.1934	0.2106	0.2134

5.3 Synchronization In A Quad-Core System With High Synchronization Overhead

The purpose of the second experiment is to show how FMLP, PWLP and FEP perform when the synchronization overhead is increased compared to case study 1. The term *high* here means that the synchronization overhead is raised as high that the system is not able to keep all of its deadlines and thus it is not robust anymore. Nevertheless, a benchmark regarding the first experiment in Section 5.2 should be possible, so that the experimental setting differ only in some details that are described in the next section. Followed by this, the results of the experiments are presented.

5.3.1 Experimental Settings

Most of the settings of this experiment are equal to the first one, thus only the differences are described in the following. Task activations are chosen randomly from range [3 ms - 33 ms]. That means, periods are shorter, but deadlines as well. Further, as the task utilization remains constant, fewer instructions per task are generated. Runnables still have the same amount of instructions: [min: 100.000; avg: 500.000; max: 1.000.000]. As a consequence, fewer runnables are available for the selection of critical sections, so that the synchronization overhead increases (nesting increases as well). Further, the risk of violating deadlines increases because they have become shorter (earlier) and only short perturbations (in terms of blocking) may have fatal effects regarding the schedulability of the task set.

5.3.2 Results

The results of this case study are presented in the same manner as the ones in Section 5.2. In contrast to the first experiments, not every task set is schedulable under application of any synchronization protocol any more. The blue line of the FEP shows that schedulability decreases faster than the one of FMLP and PWLP. From k = 0 to k = 4, the FEP still outperforms both other protocols. At k = 5, in contrast to the PWLP, the FMLP and the FEP are not able to keep all task sets schedulable (NRTmax(FMLP) = 1.10; NRTmax(FEP) = 1.11; NRTmax(PWLP)= 0.991). From k = 5 on, the NRTmax

of the FEP grows stronger than the FMLP and PWLP. All in all, the PWLP is able to keep more task sets schedulable than the FMPL or FEP and thus can be declared as the best protocol (of the observed ones) for this kind of task sets. This behavior can be seen in the boxplots of NRTmax as well. The tables 5.3 and 5.4 show again the results more in detail.

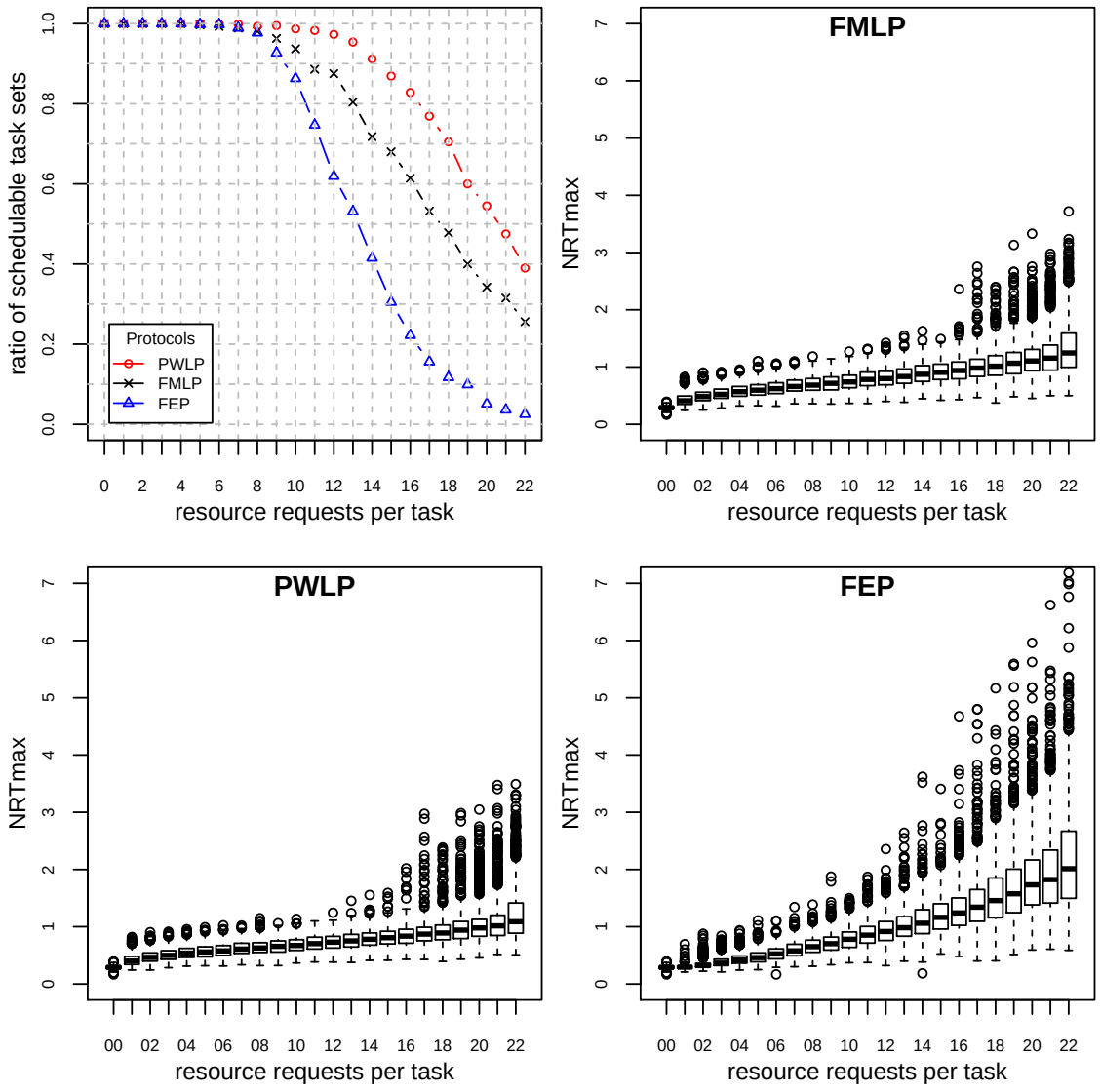

FIGURE 5.2: Results of case study 2. The upper left picture presents the ratio of schedulable task sets for all protocols. The upper right and both pictures at the bottom show the NRTmax for task sets simulated with either FMLP, PWLP and FEP respectively. The x-Axes show the resource requests per task, while the y-Axes present the ratio of schedulable task sets or rather the values of NRTmax.

TABLE 5.3: Maximum values of normalized response time of FMLP, PWLP and FEP for k = 0 - 11 in case study 2.

k	00	01	02	03	04	05	06	07	08	09	10	11
max_FMLP	0.3937	0.8276	0.9042	0.9132	0.9564	1.1003	1.0594	1.0990	1.1849	1.1427	1.2711	1.3180
max_PWLP	0.3937	0.8205	0.9042	0.9132	0.9564	0.9914	1.0244	1.0217	1.1487	1.0613	1.1374	1.1002
max_FEP	0.3937	0.6957	0.8785	0.8416	0.9372	1.1103	1.1065	1.3442	1.3880	1.8758	1.4997	1.8730
avg_FMLP	0.2904	0.4286	0.4943	0.5376	0.5816	0.6063	0.6354	0.6769	0.6997	0.7203	0.7500	0.7916
avg_PWLP	0.2904	0.4174	0.4770	0.5132	0.5501	0.5712	0.5916	0.6250	0.6421	0.6601	0.6852	0.7142
avg_FEP	0.2904	0.2952	0.3409	0.3880	0.4349	0.4760	0.5337	0.5969	0.6629	0.7199	0.7889	0.8736
min_FMLP	0.1667	0.2439	0.2491	0.2841	0.3212	0.3246	0.3172	0.3587	0.3599	0.3542	0.3659	0.3632
min_PWLP	0.1667	0.2439	0.2442	0.2841	0.3111	0.3183	0.3133	0.3347	0.3201	0.3217	0.3659	0.3841
min_FEP	0.1667	0.2090	0.2238	0.2111	0.2431	0.2499	0.1674	0.3004	0.3096	0.3360	0.3705	0.3739

TABLE 5.4: Maximum values of normalized response time of FMLP, PWLP and FEP for k = 12 - 22 in case study 2.

k	12	13	14	15	16	17	18	19	20	21	22
max_FMLP	1.4231	1.5492	1.6273	1.4893	2.3612	2.7536	2.3987	3.1330	3.3301	2.9785	3.7199
max_PWLP	1.2449	1.4529	1.5549	1.5916	2.0203	2.9736	2.3838	2.9850	3.0467	3.4762	3.4916
max_FEP	2.3566	2.6421	3.6200	3.4087	4.6768	4.8018	5.1656	8.2390	5.9586	7.9882	10.2578
avg_FMLP	0.8105	0.8442	0.8868	0.9144	0.9470	1.0013	1.0442	1.0976	1.1673	1.2315	1.3557
avg_PWLP	0.7315	0.7585	0.7882	0.8148	0.8427	0.8924	0.9250	0.9765	1.0462	1.1135	1.2421
avg_FEP	0.9421	1.0284	1.1190	1.2081	1.3108	1.4372	1.5688	1.7087	1.8734	1.9921	2.2278
min_FMLP	0.3989	0.3842	0.4468	0.4195	0.4303	0.4658	0.3721	0.4809	0.4520	0.4989	0.4997
min_PWLP	0.3821	0.3764	0.4122	0.4144	0.4300	0.4280	0.3936	0.4325	0.4554	0.5172	0.5092
min_FEP	0.3213	0.3980	0.1857	0.5246	0.4817	0.4014	0.4039	0.5153	0.5940	0.6071	0.5880

5.4 Discussion

The results of the case studies 1 and 2 are summarized in this chapter. Furthermore, a summarizing conclusion is presented.

5.4.1 Discussion of Case Study 1

In case study 1, a quadcore system with low synchronization overhead was considered in order to evaluate the contributed protocols. All protocols were able to keep all task sets schedulable. This means that all protocols can be used for real-time system when proceeding this kind of task sets. However, the FEP performs best in case study 1 concerning robustness (NRTmax).

The results for each protocol in this case study can be described as follows: The main properties of the FMLP are non-preemptive waiting and critical sections regardless of tasks priorities. This leads to the fact that under the FMLP, high priority tasks are delayed, even if there exist no concrete resource conflicts with low priority tasks that are waiting for a resource or processing a critical section meanwhile. High priority tasks under the EDF scheduling policy are that ones with the earliest relative deadline. As a consequence, delays of these early deadline tasks lead to a higher NRTmax and thus to a reduced robustness of the system. These assumptions are confirmed by the priority aware evaluation in Chapter A, which shows that the tasks that are responsible for the NRTmax of the task set are the high priority (with the earliest relative deadline) ones.

The PWLP processes critical sections in a non-preemptive manner, like the FMLP. In contrast, the busy-wait phase is preemptive. Since the term of waiting is short in case study 1, because of the moderate synchronization overhead, the difference between FMLP and PWLP is quite low. The PWLP performs slightly better than the FMLP regarding NRTmax (see Table 5.1 and Table 5.2). This can be confirmed by the evaluation of results for all task sets in Section 5.2 and in the priority aware evaluation in Section A as well.

The FEP uses blocking techniques that are different from the FMLP and the PWLP. Tasks are only blocked if 'necessary', which means that generally waiting and critical sections are performed preemptively. Resource holding jobs become priority boosted

if another job is waiting for the same resource. For systems with relatively low synchronization overhead, there is only a low probability of more than one waiting process (compared to systems with high synchronization overhead). If only one process is in wait-queue, critical section and waiting are preemptive and the scheduling scheme (here EDF) is not 'disturbed'. The results of case study 1 show that robustness can be kept nearly constant at a low level over the numbers of resource requests per task. Priority aware evaluations in Section A show that tasks with high priorities (early deadlines) have lower values of NRTmax compared to those in FMLP or PWLP. As high priority tasks are the main factor for NRTmax in case study 1, this is the reason why the FEP is able to increase robustness of systems with low synchronization overheads.

5.4.2 Discussion of Case Study 2

In contrast to case study 1, the second one provides the examination of systems with high synchronization overhead. This leads to significantly different results. First, no protocol is able to keep all task sets schedulable any more. Comparing the different protocols, the PWLP is the one that performs best over the most numbers of resource requests per tasks, since it can keep more task sets schedulable than the FMLP or the FEP. Nevertheless, up to k = 4, the FEP still produces better results regarding the NRTmax (minimum, average and maximum values). In the following, the behavior of protocols regarding case study 2 is discussed in detail.

Under the FMLP, high priority tasks are delayed because of non-preemptive waiting and critical sections (as already discussed above). Nevertheless, this behavior enables tasks to release resources as quickly as possible. This is one of the reasons why its performance regarding robustness remains at a higher level than that of the FEP for $k > 4$.

Waiting is preemptive under the PWLP, so high priority tasks are not blocked by waiting, low priority tasks. Additionally, non-preemptive critical sections enable a quick release of acquired resources. In this case study, the waiting behavior becomes more important compared to case study 2 because the probability of simultaneous resource requests grows and more jobs have to wait for resources when issuing a request. The fact that busy-wait is performed preemptively at the PWLP, leads to the result that high priority tasks (with early deadlines) are delayed less than at the FMLP. This behavior can be seen in Appendix B, Figure B.1 and Figure B.4. Because of that, the PWLP outperforms

the FEP and the FMLP for k > 4 in case study 2.

While the FEP outperforms the FMLP and the PWLP in case study 1, the experiments in case study 2 show very different results. Up to k = 4, the NRTmax of the FEP remains the lowest, which means it has the highest robustness. But when synchronization overhead continues growing, the FEP is not able to keep the deadlines, especially of high priority tasks (see Figure B.1). The reasons for that are twofold. On the one hand, at high synchronization overhead all cores may proceed priority boosted jobs. This means that their priority is higher than the base priority of tasks with the earliest deadlines that are not priority boosted. Followed by this, high priority tasks are not scheduled because the available cores are occupied by priority boosted tasks. This of course leads to delays and a reduced robustness. On the other hand, since blocking occurs indirectly (by priority boosting) and late (only if another job is waiting for the same resource), resources can not be released as quickly as under the FMLP or the PWLP. This extends waiting times for resources additionally, which also has negative consequences for the response times of tasks.

5.4.3 Conclusion

Concluding the discussion of the case studies 1 and 2, one can generally say that there exists no synchronization protocol that outperforms all other protocols for each type of task sets. The performance regarding the robustness of a task set depends mainly on the synchronization overhead. While the FEP seems to be the best protocol (of the compared ones) at low synchronization overhead, the PWLP is able to keep more task sets schedulable under high synchronization overhead. Our recommendation therefore is first to analyze the synchronization overhead of the system and choose a locking protocol afterwards when designing a software architecture for embedded multiprocessor real-time systems.

Furthermore, there are some additional notes to discuss: The presented case studies are only an excerpt of all performed experiments. We presented a case study with EDF scheduling policy because task-fix priority scheduling policies like RM and DM do not lead to significant different results [1] in the presented case studies. Further, the case studies fit better to the contributed scheduler concept. Moreover, optimal scheduling

[1] Experiments with task-fix priority scheduling were carried out as well.

approaches like Pfair and LLREF are not considered for these experiments, as they do not perform well due to high overheads (see [31]). Diverse further experiments with varied settings of task numbers, task utilization, task activation, scheduling algorithms, runnable lengths, and resource access settings have been carried out. They confirmed the results of the presented case studies that the performance of locking protocols regarding robustness depends on the task sets and there is no protocol that outperforms competing protocols. Case studies, for example performed in [15], [40] or [95], came to different results for different kinds of settings when evaluating and comparing locking protocols as well.

Chapter 6

Conclusion

This chapter gives a thesis summary, followed by a discussion of possible and necessary future works regarding the contributions of this dissertation.

6.1 Thesis Summary

The presented global scheduler for job-fix priority scheduling considers all requirements to handle task sets of practical embedded real-time systems, for example in the automotive powertrain domain. These requirements contain the integration of a buffering mechanism, (parallel performed) system-wide state transitions and further tasks with core affinities. Different types of disruption may be used, e. g. non-preemptive and cooperative. Moreover, a fast and efficient implementation is possible, as this approach can be based on an existing operating system. Further, the system is able to handle parallel and chained tasks in addition to common tasks as well as system transition tasks, and different types of activation patterns (periodic, sporadic and angle-triggered). Referring to the objectives of this thesis, described in Section 4.1, we now can conclude that objectives 1b, 1c, 1d, 1e, 1f, 1g and 1h are achieved successfully.

With the help of this approach, systems that work nowadays with partitioned allocation and task-fix priorities can be optimized in terms of robustness and efficiency. Robustness, because the scheduler is able to react dynamically on the actual workload (due to job-fix priorities) and thus it reduces the risk of deadline violations. The global approach assures that ready tasks can be scheduled on any core (except those tasks with

core affinities). In comparison, a job within a partitioned allocation may violate its deadline because it first has to wait until the running job terminates, even if other cores remain idle. Nevertheless, a mathematical analysis or practical experiments have to be performed in future work to verify that the provided algorithm is rising the robustness of the system in practice (objective 1a seems to be fulfilled, but a verification is still required). Efficiency, because a better load-balance of the system offers the opportunity to reduce the clock frequency of the processor, which leads to lower energy consumption and heat dissipation. Further, implementing efficiency is given as well, as already existing and well-tested task-fix priority schedulers can be re-used and only a few modules have to be added. Finally, an efficient implementation as described in [54] guarantees a low scheduling overhead during runtime of the system. This is a kind of efficiency as well. Nevertheless, the last efficiency term (scheduling overhead) has to be proven by implementing on real hardware or at least in simulations (see Section 6.2).

Regarding synchronization, we contributed two different locking protocols, the preemptable waiting protocol PWLP and the forced execution protocol FEP. Both protocols have FIFO ordered wait-queues for resource requests and support nesting as well as all types of allocation and prioritization. This means, that the thesis objectives 2a and 2b are reached. The PWLP is based on non-preemptive critical sections, whereas busy-wait is performed preemptively. Compared to a benchmark protocol, the short variant of the FMLP, priority inversions are reduced, since high priority tasks are able to preempt busy-wait phases of low priority tasks. Nevertheless, the robustness of the PWLP depends on the task set parameters.

In contrast to the PWLP, the FEP additionally allows critical sections to be preempted. Blocking occurs only if necessary, namely when at least one task's resource request is denied. Resource holding tasks are forced to execute by a priority boosting mechanism. The presented case studies show that the performance regarding robustness of the FEP is reduced if nesting is applied and long chains of synchronization occur.

Generally, the results of the case studies in Chapter 5 show, that there exists no locking protocol that outperforms all the other solutions in every case. Therefore, a new decision for a specific locking protocol has to be made for every change of the system (task set, hardware,...) separately. Nevertheless, both provided protocols, the FEP and the PWLP, are able to make synchronization more efficient for certain types of task sets, which results in a higher robustness of those systems. This means that the thesis

objective 2c is reached as well.

In conclusion, the goal of this thesis to rise efficiency and robustness of embedded real-time multiprocessor systems is reached, as well as moving a step forward in terms of closing the gap between scheduling and synchronization theory and practical systems.

6.2 Future Work

In this thesis, we developed a concept of a scheduler with global allocation and job-fix priorities. However, a hardware implementation of this scheduler on an engine control unit ECU in an automotive powertrain system still has to be realized. An efficient implementation concept has already been developed by [54]. As a first step, the scheduling algorithm itself has to be realized, followed by the integration of system state transitions, buffering and the concept of parallel and chained tasks. We further recommend an extension of the event-based simulator, so that this algorithm can be described properly and simulation experiments can be performed, for instance in combination with different scheduling policies. This leads to the next step, the adaption of further scheduling algorithms into the concept of this thesis, for example the P-ERfair-PD^2 algorithm. Additionally, solutions for clustered allocation are required when the number of cores will increase further and, as a result, one single global job-list would be accompanied by too high scheduling overheads.

Further work regarding locking protocols may contain a detailed evaluation of overheads for semaphore accesses, task migration and context switches. Therefore, a traceable hardware implementation would be desirable. Especially a 'real world' case study, for example with an engine control unit task set from the automotive powertrain domain, would be interesting. Experiments with other resource models, like phase-fair reader-writer locks or k-replica would be promising as well. Finally, non-blocking synchronization techniques like transactional memory should be observed regarding their suitability for embedded real-time systems, as the hardware may become capable for such mechanisms in the future. Nevertheless, it should be mentioned that non-blocking synchronization is accompanied by overhead as well. In case of using replica, memory overhead is required. In the opposite, when algorithms with restarting calculations are used, runtime overhead increases. Both cases have negative impact on the system, as

embedded technology provides only as few memory resources as necessary, and real-time systems require low timing overheads in order to fulfill their temporal constraints.

Appendix A

Priority Aware Evaluation Of Case Study 1

Figure A.1 presents a priority aware evaluation of the experiment in case study 1 (see Section 5.2). Each simulated task set is divided into three values of priorities, namely high-priority, medium-priority and low-priority tasks. Since EDF is used as scheduling scheme, tasks are sorted by relative deadline, where the 10 tasks with the earliest deadlines (each task set contains 30 tasks) are denoted to have the highest priorities.

Followed by this, the protocols can be analyzed if the NRTmax of a task set is caused by either high-priority, medium-priority or low-priority tasks. This kind of observation is mainly interesting for systems with mixed criticality, where high-priority (or short deadline) tasks have hard real-time requirements, whereas low priority (or long deadline) tasks only have firm or even soft real-time requirements.

However, the priority aware evaluation of the experiment in case study 1 show that the main difference of the applied protocols lies in the NRTmax of high-priority tasks, where the NRTmax values of the FEP are below the values of FMLP and PWLP. The reason for that is that under the FEP, critical sections and wait-phases are preemptable, so that high-priority tasks are allowed to execute although another task is running with acquired resources. In contrast, under PWLP only waiting is preemptable and under FMLP, waiting and critical sections are both non-preemptable for any task.

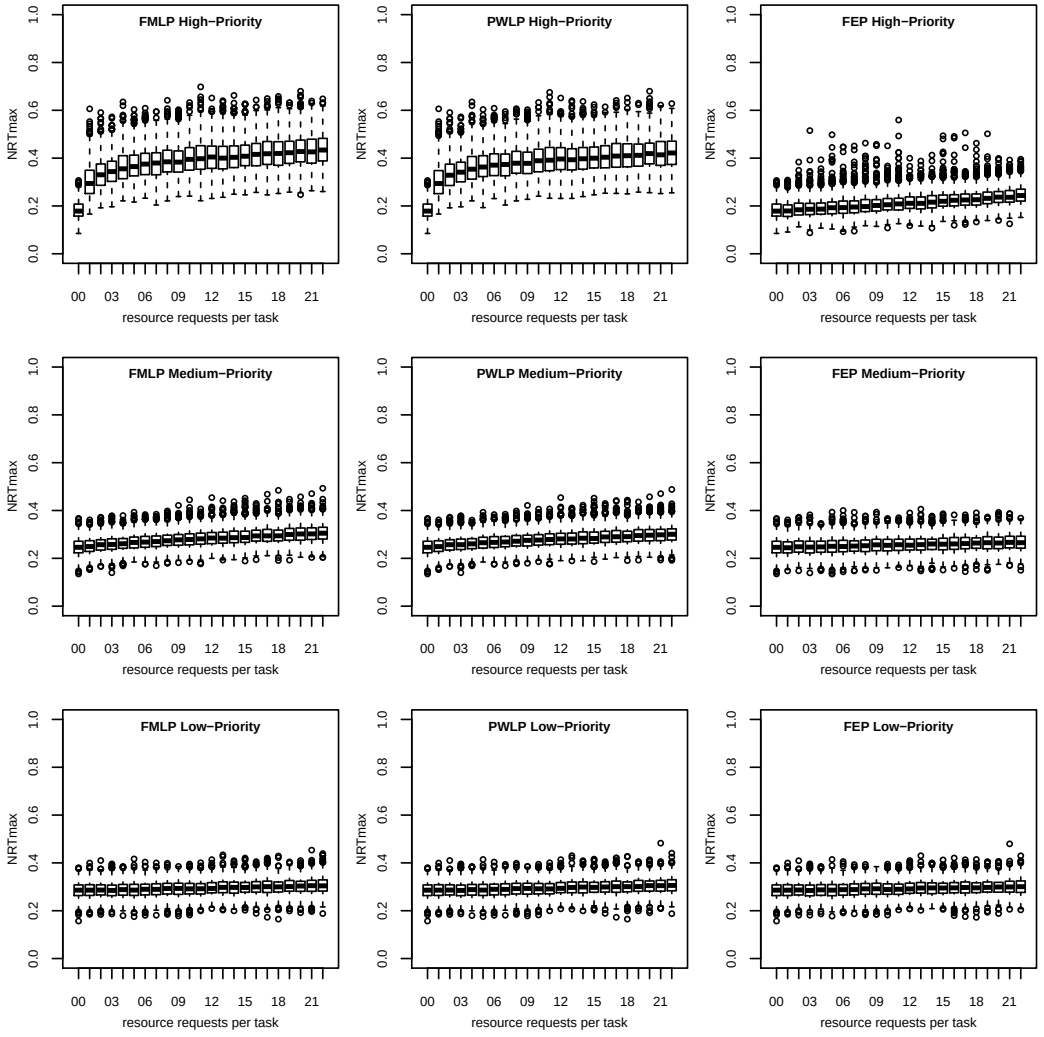

FIGURE A.1: Results of Experiment 1. Schedulable task sets filtered by low-priority tasks. The x-Axes show the resource requests per task, while the y-Axes present the values of NRTmax.

Appendix B

Priority Aware Evaluation Of Case Study 2

This appendix presents a priority aware evaluation of case study 2, which was presented in Section 5.3. The division of the tasks into groups happens similar to that in Section A. In contrast to the first case study, not all task sets remain schedulable. Followed by this, Figures B.1, B.2 and B.3 show the ratio of schedulable task sets of high-priority, medium-priority and low-priority tasks.

One can observe that similar to case study 1, mainly the high priority tasks cause deadline violation and thus prevent the task sets from schedulability. Further, the boxplot in Figure B.4 shows the same evaluation like in Section A. This figure shows also that high-priority tasks lead to a higher level of NRTmax.

In contrast to case study 1, the FEP performs worst here because the available cores are blocked frequently by priority boosted tasks. This happens if more than one task has sent a resource request for a resource. Followed by this, high-priority tasks that have not yet acquired a resource and thus are not priority boosted, do not have the opportunity to be processed and are liable to violate their deadlines. In contrast, FMLP and PWLP have non-preemptive critical sections, which means that resources are released earlier when acquired. As a result, high priority tasks can be scheduled directly after the currently running low priority task releases its resource. Nevertheless, the property of PWLP, that high-priority tasks are able to preempt waiting low priority tasks, leads to a higher schedulability of PWLP compared to FMLP.

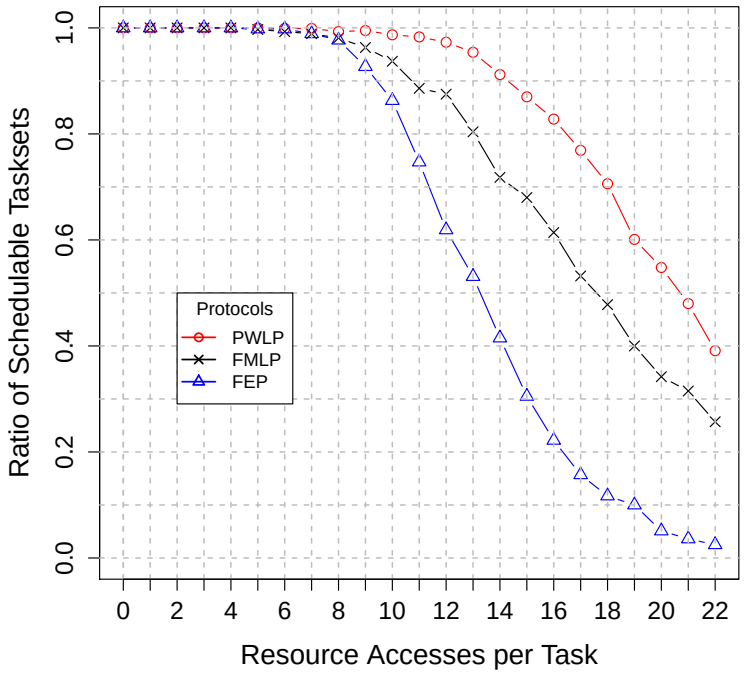

FIGURE B.1: Results of Experiment 2. Schedulable task sets filtered by high-priority tasks. The x-Axes show the resource requests per task, while the y-Axes present the ratio of schedulable task sets.

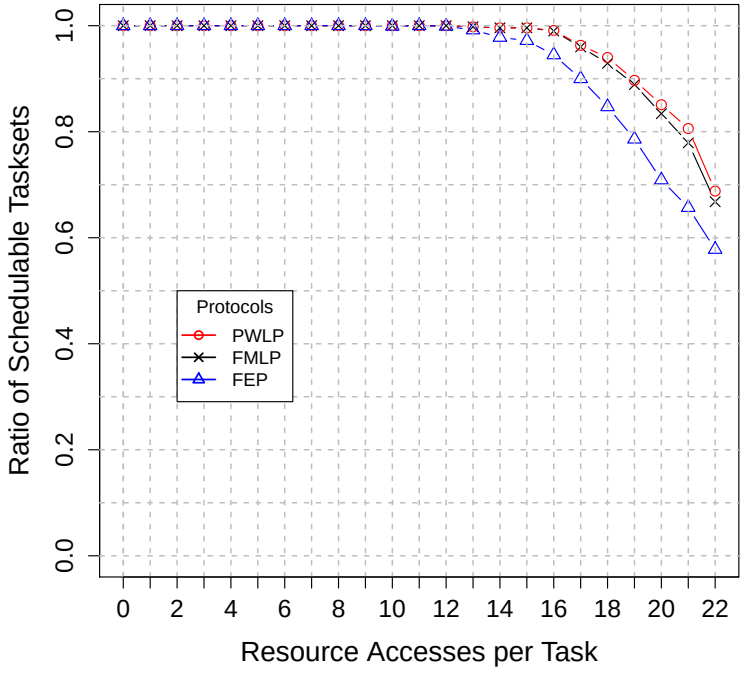

FIGURE B.2: Results of Experiment 2. Schedulable task sets filtered by medium-priority tasks. The x-Axes show the resource requests per task, while the y-Axes present the ratio of schedulable task sets.

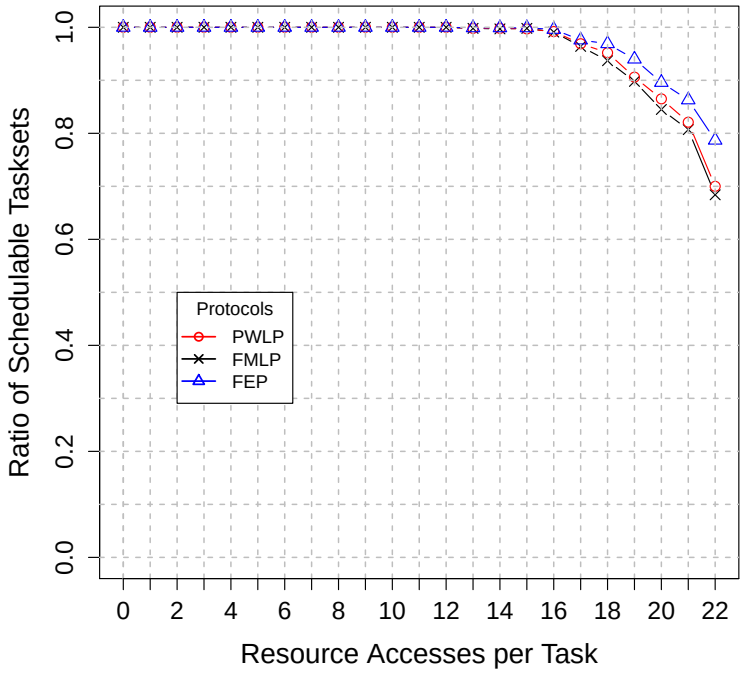

FIGURE B.3: Results of Experiment 2. Schedulable task sets filtered by low-priority tasks. The x-Axes show the resource requests per task, while the y-Axes present the ratio of schedulable task sets.

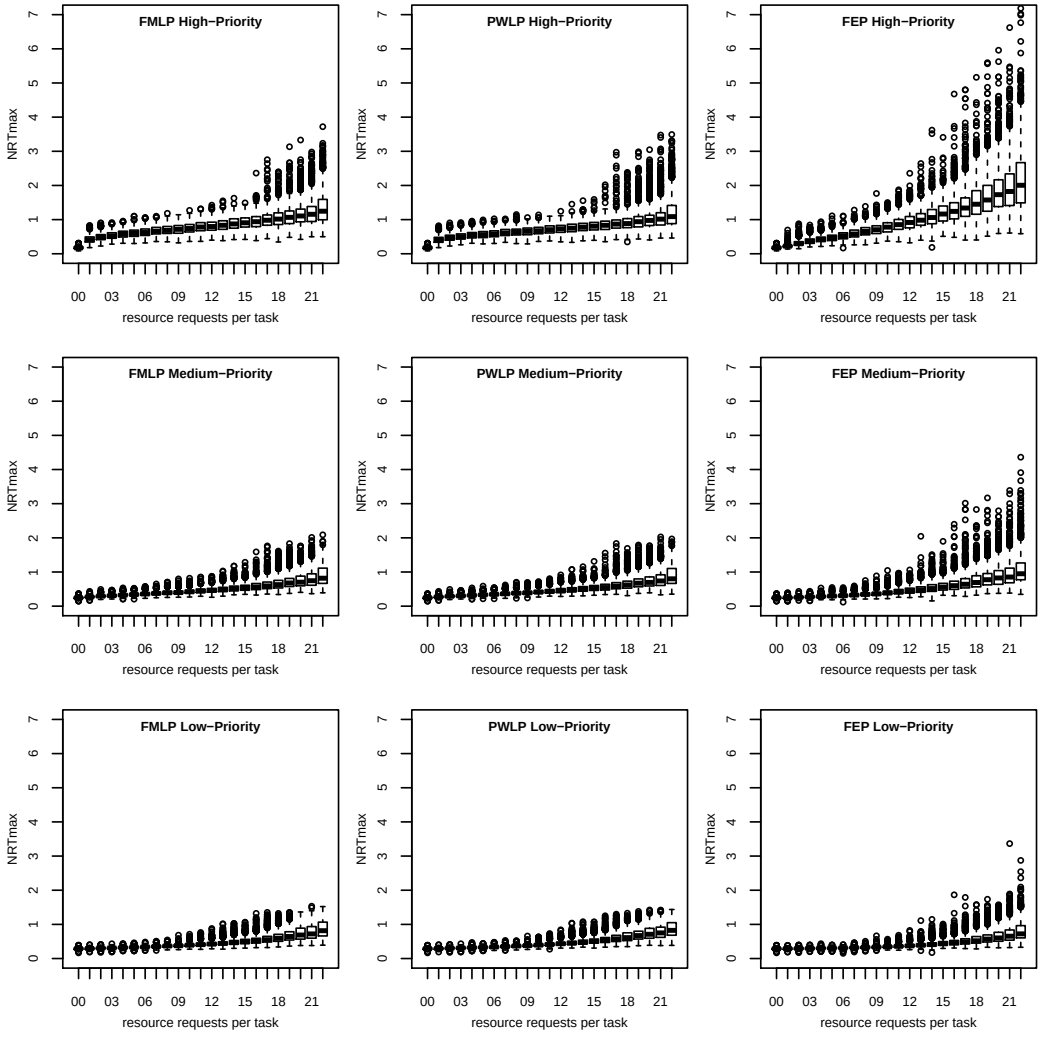

FIGURE B.4: Results of Experiment 2. Schedulable task sets filtered by low-priority tasks. The x-Axes show the resource requests per task, while the y-Axes present values of NRTmax.

Bibliography

[1] Continental AG. Core synchronous transitions in embedded software for multi cores, 2013.

[2] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers. An efficient spin-lock based multi-core resource sharing protocol. In *In Proceedings of the 33rd IEEE International Performance, Computing, and Communication Conference (IPCCC)*, 2014.

[3] M. Alfranseder, T. Krapf, R. Mader, M. Niemetz, J. Mottok, and C. Siemers. An efficient partitioning strategy for runnables in weakly dependent tasks on embedded multi-core systems. In *Embedded Realtime Systems and Software Congress, Toulouse*, 2014.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. ISSN 0304-3975.

[5] James H. Anderson and Anand Srinivasan. Early-release fair scheduling. In *In Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000.

[6] James H. Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ECRTS '01, pages 76–, Washington, DC, USA, 2001. IEEE Computer Society.

[7] James H. Anderson, Rohit Jain, and Kevin Jeffay. Efficient object sharing in quantum-based real-time systems - summary by andreas haeberlen. In *In Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. IEEE, 1998.

[8] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA '00, pages 337–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0930-4.

[9] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 322–334, 2006.

[10] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 33–33. IEEE Computer Society, 2003.

[11] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*, RTSS '01, pages 93–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1420-0.

[12] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[13] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Technical Report YCS 164*, Dept. Computer Science, University of York, UK, 1991.

[14] AUTOSAR. *AUTomotive Open System ARchitecture Release 4.1*, 2013.

[15] B. B. Brandenburg and James H. Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmusrt. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 105–124, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-92220-9.

[16] T. P. Baker. A stack-based resource allocation policy for realtime processes. *In proceeding of the 11th Real-Time Systems Symposium*, pages 191–200, 1990.

[17] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[18] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 119–128, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3062-1.

[19] Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 159–168, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2761-2.

[20] Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity. *Handbook of scheduling: Algorithms, models, and performance analysis*, 3, 2004.

[21] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, International*, pages 280–280. IEEE Computer Society, 1995.

[22] Jan Beirlant, Yuri Goegebeur, Jozef Teugels, and Johan Segers. *Statistics of Extremes: Theory and Applications*. Wiley series in probability and statistics. John Wiley & Sons, Ltd, 2005.

[23] Antoine Bertout, Julien Forget, and Richard Olejnik. Research report- automated runnable to task mapping, 2013.

[24] A. Biondi, A. Melani, M. Marinoni, M. Di Natale, and G. Buttazzo. Exact interference of adaptive variable-rate tasks under fixed-priority scheduling. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 165–174, July 2014.

[25] Jacek Blazewicz, Klaus H. Ecker, Gnter Schmidt, and Jan Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer Publishing Company, Incorporated, 2nd edition, 1993. ISBN 3540559582, 9783540559580.

[26] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 0:47–56, 2007. ISSN 1533-2306.

[27] B. Brandenburg and J. H. Anderson. The omlp family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 2012. ISSN 0929-5585.

[28] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 69–78, Oct 2011.

[29] B.B. Brandenburg, J.M. Calandrino, and J.H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium, 2008*, pages 157–169, Nov 2008.

[30] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 342–353, April 2008.

[31] Björn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

[32] Björn B Brandenburg. An asymptotically optimal real-time locking protocol for clustered scheduling under suspension-aware analysis. *SIGBED Review*, 10(2):19, 2013.

[33] Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems, 2009.

[34] A Burns and AJ. Wellings. A schedulability compatible multiprocessor resource sharing protocol – mrsp. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291, July 2013.

[35] Alan Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[36] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. ISBN 0387231374.

[37] Giorgio C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Syst.*, 29 (1):5–26, January 2005. ISSN 0922-6443.

[38] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer, 2011.

[39] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid realtime scheduling approach for large-scale multicore platforms, 2007.

[40] Andreu Carminati, Rômulo Silva De Oliveira, and Luís Fernando Friedrich. Exploring the design space of multiprocessor synchronization protocols for real-time systems. *J. Syst. Archit.*, 60(3):258–270, March 2014. ISSN 1383-7621.

[41] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.

[42] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback–feedforward scheduling of control tasks. *Real-Time Systems*, 23(1-2):25–53, 2002.

[43] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10190–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2.

[44] Younes Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms. In *WATERS 2012*, pages 21–26. UPE LIGM ESIEE, 2012.

[45] Chia-Mei Chen and Satish K. Tripathi. Multiprocessor priority ceiling based protocols. Technical report, Univ. of Maryland, 1994.

[46] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–p, 2014.

[47] Hyeonjoong Cho and Binoy Ravindran. An optimal realtime scheduling algorithm for multiprocessors. In *In Proc. 27th IEEE International Real-Time Systems Symposium, Rio de Janeiro*, pages 101–110, 2006.

[48] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In *Handbook of Automated Reasoning*, volume 2, pages 1637–1790. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2011.

[49] D. Compagnin, E. Mezzetti, and T. Vardanega. Putting run into practice: Implementation and evaluation. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 75–84, July 2014.

[50] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Kepler tm gk110/210. Whitepaper, 2014.

[51] NVIDIA Corporation. Gpu accelerated applications. Catalog, 2014.

[52] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, October 1971. ISSN 0001-0782.

[53] T.S. Craig. Queuing spin lock algorithms to support timing predictability. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 148–157, Dec 1993.

[54] Markus Daub. Konzeption, entwurf und realisierung sowie simulation und validierung eines edf schedulers, um eine verbrennungsmotorregelung auf multicore plattformen optimal auszuführen. Bachelor Thesis, 2015.

[55] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. ISSN 0360-0300.

[56] Luca De Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 148–165, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42673-6.

[57] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. Softw. Eng.*, 15(12):1497–1506, December 1989. ISSN 0098-5589.

[58] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[59] M. Deubzer. *Robust Scheduling of Real-Time Applications on Efficient Embedded Multicore Systems*. PhD thesis, Technische Universität, München, 2011.

[60] M. Deubzer, F. Schiller, J. Mottok, M. Niemetz, and U. Margull. Effizientes multicore-scheduling in eingebetteten systemen - teil 2: Ein simulationsbasierter ansatz zum vergleich von scheduling-algorithmen. *atp - Automatisierungstechnische Praxis*, pages 54–63, October 2010.

[61] Michael Deubzer, Ulrich Margull, Jürgen Mottok, Michael Niemetz, and Gerhard Wirrer. Partly proportionate fair multiprocessor scheduling of heterogeneous task systems. *Proceedings of the 5th Embedded Real Time Software and Systems Conference*, 2010.

[62] Michael Deubzer, J Mottok, U Margull, and Michael Niemetz. Efficient scheduling of reliable automotive multi-core systems with pd2 by weakening pfair tasksystem requirements. *Proceedings of the Automotive Safety & Security*, 2010.

[63] Umamaheswari C Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[64] Umamaheswari C. Devi, Hennadiy Leontyev, North Carolina, Chapel Hill, and Abstract We. Efficient synchronization under global edf scheduling on multiprocessors. In *In Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 75–84, 2006.

[65] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[66] Claas Diederichs, Ulrich Margull, Frank Slomka, and Gerhard Wirrer. An application-based edf scheduler for osek/vdx. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1045–1050, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1.

[67] J.P. Erickson and J.H. Anderson. Outstanding paper award: Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 3–12, July 2012.

[68] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. 22nd Euromicro Conference on Real-Time Systems (ECRTS), IEEE, 2010. ISBN 978-1-4244-7546-9.

[69] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Syst.*, 45(1-2): 26–71, June 2010. ISSN 0922-6443.

[70] K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 13–22, July 2008.

[71] Shelby Funk. Lre-tl: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, 46(3):332–359, December 2010. ISSN 0922-6443.

[72] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *In Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83. Society Press, 2001.

[73] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 189–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1956-3.

[74] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[75] Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 63–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1.

[76] Continental Automotive GmbH. De-102007042999: Edf-implementierung für realzeitsysteme mit statischen prioritäten, 2009.

[77] Continental Automotive GmbH. Anmeldekennzeichen de-102015218431.5: Edf-implementierung für realzeitsysteme mit statischen prioritäten, 2015.

[78] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, September 2003. ISSN 0922-6443.

[79] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 263–272. Springer, 2007.

[80] M González Harbour, Mark H Klein, and John P Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 116–128. IEEE, 1991.

[81] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *In 13th Euromicro Conference on Real-Time Systems*, page 125, 2001.

[82] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[83] M. Holenderski, R. Bril, and J. Lukkien. Parallel-task scheduling on multiple resources. In *Proceedings of the 24rd 24th Euromicro Conference on Real-Time Systems*, pages 233–244, Washington, DC, USA, 2012. IEEE Computer Society.

[84] Philip Holman and James H. Anderson. Locking under pfair scheduling. In *In ACM Transactions on Computer Systems*, pages 140–174, 2006.

[85] Mathai Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[86] Kalray. Mppa®: The supercomputing on a chip™ solution. [Online; accessed on 26-June-2015]. URL `http://www.kalrayinc.com/kalray/products/#processors`.

[87] Cem Kaner and Walter P Bond. Software engineering metrics: What do they measure and how do we know? *methodology*, 8:6, 2004.

[88] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23–32, April 2009.

[89] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 139–148. ACM, 2008.

[90] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-time Analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0-7923-9361-9.

[91] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, February 1997. ISSN 0734-2071.

[92] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997. ISBN 0792398947.

[93] Hermann Kopetz. Wrong assumptions and neglected areas in embedded systems research. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 360–360, May 2008.

[94] K. Lakshmanan, R. Rajkumar, and J.P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 239–248, July 2009.

[95] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 469–478, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3875-4.

[96] Gerardo Lamastra, Giuseppe Lipari, Telecom Italia Lab, Luca Abeni, and Scuola Superiore S. Anna. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *In Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 151–160, 2001.

[97] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001. ISBN 3-540-42184-X.

[98] H. Leontyev and J.H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 191–200, July 2008.

[99] H. Leontyev, S. Chakraborty, and J.H. Anderson. Multiprocessor extensions to real-time calculus. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 410–421, Dec 2009.

[100] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[101] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411.

[102] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. ISBN 0130996513.

[103] J. M. López, M. García, J. L. Díaz, and D. F. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, Euromicro-RTS'00, pages 25–33, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0734-4.

[104] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, October 2004. ISSN 0922-6443.

[105] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):642–653, July 2004. ISSN 1045-9219.

[106] Yue Lu, Thomas Nolte, Johan Kraft, and Christer Norstrom. A statistical approach to response-time analysis of complex embedded real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 153–160. IEEE, 2010.

[107] G. Macariu and V. Cretu. Timed automata model for component-based real-time systems. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 121–130, March 2010.

[108] Jan Madsen, Michael R Hansen, Kristian S Knudsen, Jens E Nielsen, and Aske W Brekling. System-level verification of multi-core embedded systems using timed-automata. In *Proceedings of the 17th World Congress International Federation of Automatic Control Seoul, Korea*, pages 9302–9307, 2008.

[109] Ulrich Margull, Michael Niemetz, and Gerhard Wirrer. Quirks and challenges in the design and verification of efficient, high-load real-time software systems. *5th Embedded Real Time Software and Systems Conference*, 2010.

[110] A. K. Mok. *Fundamental design problems of distributed systems for the hard real-time environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[111] Farhang Nemati and Thomas Nolte. Resource sharing among prioritized real-time applications on multi-cores. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-265/2012-1-SE, April 2012.

[112] Farhang Nemati and Thomas Nolte. Resource sharing among real-time components under multiprocessor clustered scheduling. *Real-Time Systems*, 49(5):580–613, 2013.

[113] Farhang Nemati, Moris Behnam, and Thomas Nolte. Independently-developed real-time systems on multi-cores with shared resources. *2012 24th Euromicro Conference on Real-Time Systems*, 0:251–261, 2011. ISSN 1068-3070.

[114] D. Oh and T.P. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15:183–192, 1998.

[115] S.-H. Oh and S.-M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, RTCSA '98, pages 31–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-9209-X.

[116] OSEK. *OSEK/VDX Operating System Specification 2.2.3*, 2005.

[117] L.T.X. Phan, S. Chakraborty, P.S. Thiagarajan, and L. Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 343–352, Dec 2007.

[118] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123, May 1990.

[119] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991. ISBN 0792392116.

[120] R. Rajkumar, Lui Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269, Dec 1988.

[121] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 104–115. IEEE, 2011.

[122] Soheil Samii, Sergiu Rafiliu, Petru Eles, and Zebo Peng. A simulation methodology for worst-case response time estimation of distributed real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 556–561, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1.

[123] Luca Santinelli and Liliana Cucu-Grosjean. Toward probabilistic real-time calculus. *SIGBED Rev.*, 8(1):54–61, March 2011. ISSN 1551-3688.

[124] Freescale Semiconductor. Mpc5676r microcontroller datasheet, December 2014.

[125] L. Sha, R. Rajkumar, and J.P Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, 1990. ISSN 00189340.

[126] Lui Sha, Tarek Abdelzaher, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004. ISSN 0922-6443.

[127] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 181–190, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3298-1.

[128] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: A flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-time &Amp; Distributed Systems Using Ada and Related Technologies*, SIGAda '04, pages 1–8, New York, NY, USA, 2004. ACM. ISBN 1-58113-906-3.

[129] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93 – 98, 2002. ISSN 0020-0190.

[130] H. Takada and K. Sakamura. Predictable spin lock algorithms with preemption. In *Real-Time Operating Systems and Software, 1994. RTOSS '94, Proceedings., 11th IEEE Workshop on*, pages 2–6, May 1994.

[131] Hiroaki Takada, Hiroaki Takada, Ken Sakamura, and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels, 1997.

[132] A.S. Tanenbaum. *Computerarchitektur: Strukturen, Konzepte, Grundlagen.* Informatik : Rechnerarchitektur. Pearson Studium, 2006. ISBN 9783827371515.

[133] Infineon Technologies. Highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications, 2015.

[134] Tilera. Tile-mx processors. [Online; accessed on 26-June-2015]. URL `http://http://www.tilera.com/`.

[135] Timing-Architects. TA Academic & Research License Program, Simulator V14.03.1, 2014. URL `http://www.timing-architects.com`.

[136] R. Urunuela, A. De Planche, and Y. Trinquet. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8, Sept 2010.

[137] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24rd 24th Euromicro Conference on Real-Time Systems*, pages 223–232, Washington, DC, USA, 2012. IEEE Computer Society.

[138] A. Wieder and B.B. Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 45–56, Dec 2013.

[139] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 351–360, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3787-0.

[140] D. Zhu, D. Mosse, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 142–151, Dec 2003.